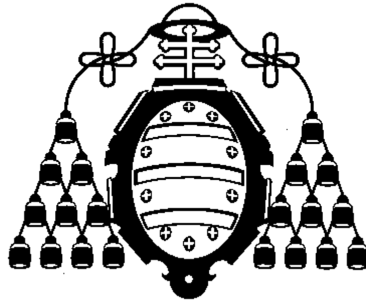
UNIVERSIDAD DE OVIEDO

Departamento de Informática



TESIS DOCTORAL

AGRA: Sistema de Distribución de Objetos para un Sistema Distribuido Orientado a Objetos soportado por una Máquina Abstracta

Presentada por

Fernando Álvarez García

para la obtención del título de Doctor Ingeniero en Informática

Dirigida por el

Profesor Doctor D. Juan Manuel Cueva Lovelle

Resumen

Esta Tesis describe la forma en que se pueden utilizar las tecnologías orientadas a objetos para la construcción de un sistema operativo distribuido que se convertirá, en definitiva, en un sistema operativo distribuido orientado a objetos, de tal forma que se conjuguen las ventajas de ambas aproximaciones.

En sistemas operativos actuales que se definen como distribuidos y orientados a objetos, la aplicación de las herramientas de distribución y de orientación a objetos se realiza de una manera parcial, dado que se sustentan en los conceptos tradicionales de posición de memoria, proceso y fichero. Además, los mecanismos y políticas definidos por el sistema operativo están codificados de manera rígida en el mismo, y no es posible adaptarlos al entorno en el que va a funcionar.

Una aproximación interesante a la integración de ambas tecnologías, que es la presentada en este trabajo, es la creación de un sistema que ofrezca un soporte directo y común para el paradigma de la orientación a objetos, obteniéndose así un sistema integral orientado a objetos. En este sistema, todos los elementos, interfaces de usuario, aplicaciones, lenguajes, etc., y el mismo sistema operativo comparten el mismo paradigma de la orientación a objetos.

La arquitectura del sistema integral está basada en una máquina abstracta orientada a objetos, que ofrece una funcionalidad mínima, y un sistema operativo orientado a objetos, que extiende la funcionalidad de la máquina. La distribución de objetos, aspecto central de esta Tesis, forma parte de la funcionalidad del sistema operativo que no estaba presente en la máquina abstracta.

La integración de la máquina abstracta y el sistema operativo se consigue a través de la arquitectura reflectiva de la primera, que permite definir la máquina como objetos normales. El sistema operativo extiende uniformemente la funcionalidad de la máquina definiendo un conjunto de objetos que sustituyen a los objetos internos de aquella.

El sistema integral se convierte en un sistema integral distribuido en el momento en que el sistema operativo implementa la característica de distribución. Existirán, entonces, varias máquinas abstractas que cooperarán gracias al sistema operativo. La extensión realizada por el sistema operativo para la distribución es llevada a cabo en dos sentidos: en primer lugar, se posibilita la invocación de objetos de manera transparente con independencia de su ubicación; en segundo lugar, los objetos pueden moverse entre las diferentes máquinas abstractas del sistema.

Con el fin de proporcionar las dos facilidades de distribución mencionadas, se definen un conjunto de objetos del sistema operativo (meta-objetos) asociados a cada objeto, que describen un comportamiento que sustituye al descrito de manera primitiva por la máquina abstracta.

El resultado es un sistema flexible, que permite al diseñador del sistema su personalización, sustituyendo los objetos del sistema operativo, y con una aplicación completa de las tecnologías de orientación a objetos.

Palabras clave

Orientación a objetos, tecnologías orientadas a objetos, sistema integral orientado a objetos, máquinas abstractas, sistemas distribuidos, sistemas operativos, sistemas operativos distribuidos orientados a objetos, flexibilidad, uniformidad, meta-objeto, invocación remota, migración de objetos, reflectividad.

Abstract

This Thesis describes how object-oriented technologies can be used with the aim of building a distributed operating system. The resulted system, a distributed object-oriented operating system, will combine the advantages of both approximations.

In present operating systems, defined as distributed an object-oriented, both technologies are partially applied, due to the use of traditional concepts such as memory address, process and file. Furthermore, mechanisms and policies defined by the operating system are rigidly implemented, so making no possible to adapt them for a given environment.

An interesting approach, that is researched in this Thesis, for the integration of both technologies is to build a system offering direct and common support for the object-oriented paradigm to create an integral object-oriented system. In this system, all the components, user interfaces, applications, languages, etc., even the operating system itself, share the same object-oriented paradigm.

The architecture of the integral system is based on an object-oriented abstract machine, that provides the basic functionality, and an object-oriented operating system, that extends the machine functionality. Object distribution, main issue of this Thesis, is part of the functionality not present in the abstract machine.

The integration of the abstract machine and the operating system is achieved by means of the reflective architecture of the former. Reflection allows the definition of the machine as normal objects. The operating system extends uniformly the machine functionality by defining a set of objects that replace inner machine ones.

The integral system moves to a distributed one when the operating system implements the distribution feature. Several abstract machines will cooperate thanks to the operating system. The operating system extension is done in two directions: first of all, objects can be invoked transparently without regard of their location; in second place, objects can migrate between machines of the integral system.

With the aim of providing the two mentioned distribution facilities, a set of operating system objects is defined (meta-objects) and associated to every object. Those meta-objects describe a behavior that substitutes the primitive one described by the abstract machine.

The result is a flexible system that permits system designers to customize them, substituting the operating system objects, and with a complete application of the object-oriented technologies.

Palabras clave

Object-orientation, object-oriented technologies, object-oriented integral system, abstract machine, distributed system, operating system, distributed object-oriented operating system, flexibility, uniformity, meta-object, remote invocation, object migration, reflection.

Tabla de contenidos

CAPÍTULO 1 INTRODUCCIÓN, OBJETIVOS Y ORGANIZACIÓN DE LA TESIS 1

1.1 PROBLEMAS DE LA UTILIZACIÓN CONVENCIONAL DE LAS TECNOLOGÍAS ORIENTADAS A OBJETOS PARA LA CONSTRUCCIÓN DE SISTEMAS DISTRIBUIDOS 1
1.1.1 El problema de la desadaptación de impedancias o salto semántico 1
1.1.2 El problema de la interoperabilidad entre modelos de objetos........ 5
1.1.3 Problemas de las capas de adaptación sobre sistemas tradicionales 6
1.2 OBJETIVOS........ 7
1.2.1 El objeto como abstracción única........ 7
1.2.2 Modelo de objetos uniforme 8
1.2.3 Transparencia 8
1.2.4 Migración de grano fino........ 8
1.2.5 Flexibilidad, extensibilidad, adaptabilidad........ 8
1.2.6 Independencia entre mecanismos........ 9
1.2.7 Independencia de mecanismos y políticas........ 9
1.3 PROYECTO DE INVESTIGACIÓN EN EL QUE SE ENMARCA LA TESIS........ 9
1.4 ORGANIZACIÓN DE LOS CONTENIDOS........ 9
1.4.1 Antecedentes 9
1.4.2 Solución........ 10
1.4.3 Prototipo........ 11
1.4.4 Conclusiones........ 11

CAPÍTULO 2 CONCEPTOS DE SISTEMAS DISTRIBUIDOS 13

2.1 INTRODUCCIÓN........ 13
2.2 ¿QUÉ ES UN SISTEMA DISTRIBUIDO?........ 13
2.3 CARACTERÍSTICAS CLAVE DE LOS SISTEMAS DISTRIBUIDOS........ 14
2.3.1 Compartición de recursos........ 14
2.3.2 Apertura (del inglés, openness) 15
2.3.3 Flexibilidad 16
2.3.4 Concurrencia........ 17
2.3.5 Rendimiento 19
2.3.6 Escalabilidad........ 19
2.3.7 Fiabilidad y tolerancia a fallos........ 20
2.3.8 Disponibilidad........ 22
2.3.9 Transparencia 23
2.3.10 Heterogeneidad........ 26
2.3.11 Seguridad........ 26
2.4 VENTAJAS E INCONVENIENTES DE LOS SISTEMAS DISTRIBUIDOS 27
2.4.1 Ventajas de los sistemas distribuidos frente a los centralizados........ 27
2.4.2 Desventajas de los sistemas distribuidos........ 28
2.5 DEFINICIÓN Y FUNCIONES DE UN SISTEMA OPERATIVO DISTRIBUIDO 28
2.6 RESUMEN........ 30

CAPÍTULO 3 REQUISITOS DE UN SISTEMA DE DISTRIBUCIÓN DE OBJETOS 31

3.1 INTRODUCCIÓN........ 31
3.2 ARQUITECTURA DE SISTEMAS DISTRIBUIDOS........ 31
3.2.1 Modelo de procesos o cliente/servidor........ 31
3.2.2 Un modelo intermedio........ 32
3.2.3 Modelo basado en objetos........ 33
3.3 MODELO DE OBJETOS........ 34
3.3.1 Características del modelo de objetos de Booch 34
3.3.2 Características adicionales necesarias........ 37
3.4 APLICACIÓN DE LAS TECNOLOGÍAS DE OBJETOS A LA CONSTRUCCIÓN DE SISTEMAS DISTRIBUIDOS.. 38
3.4.1 Diseño del sistema distribuido como un marco orientado a objetos........ 39
3.4.2 Diseño del sistema distribuido como soporte de objetos........ 40

3.5 CUESTIONES DE DISEÑO DEL SOPORTE DE OBJETOS EN EL SISTEMA OPERATIVO DISTRIBUIDO 40
3.5.1 Actividad interna *40*
3.5.2 Granularidad de los objetos *42*
3.5.3 Visibilidad: objetos privados/objetos compartidos *43*
3.5.4 Objetos volátiles/objetos persistentes *43*
3.5.5 Movilidad: objetos fijos/objetos móviles *44*
3.5.6 Replicación: objetos de una copia/ objetos replicados *44*
3.5.7 ¿Qué es, entonces, un objeto distribuido? *45*
3.6 PROBLEMAS DE LA GESTIÓN DE OBJETOS 45
3.6.1 Nombrado *45*
3.6.2 Localización *46*
3.6.3 Acceso *46*
3.6.4 Compartición y protección *46*
3.6.5 Persistencia *46*
3.7 CARACTERÍSTICAS DE UN SISTEMA OPERATIVO DISTRIBUIDO ORIENTADO A OBJETOS 47
3.8 RESUMEN 48

CAPÍTULO 4 PANORÁMICA DE SISTEMAS DISTRIBUIDOS ORIENTADOS A OBJETOS... 49

4.1 INTRODUCCIÓN 49
4.2 DCE Y DC++ 49
4.2.1 Modelo de programación *49*
4.2.2 RPC *49*
4.2.3 Hilos *50*
4.2.4 Formato de los datos *50*
4.2.5 DC++ *50*
4.2.6 Crítica *51*
4.2.7 Características interesantes *51*
4.3 CORBA 52
4.3.1 Objetos CORBA *52*
4.3.2 El ORB *52*
4.3.3 Definición de objetos CORBA *53*
4.3.4 El repositorio de interfaces *53*
4.3.5 La Interfaz de Invocación Dinámica *54*
4.3.6 Adaptadores de objetos *54*
4.3.7 Referencias *54*
4.3.8 Paso de parámetros por valor *54*
4.3.9 Crítica *55*
4.3.10 Características interesantes *55*
4.4 DCOM 56
4.4.1 Modelo de objetos *56*
4.4.2 Interoperabilidad entre objetos COM *56*
4.4.3 Modelo de programación *57*
4.4.4 Ciclo de vida *58*
4.4.5 COM+ *58*
4.4.6 Crítica *59*
4.4.7 Características interesantes *59*
4.5 RMI DE JAVA 59
4.5.1 Objetos RMI *60*
4.5.2 Modelo de programación *60*
4.5.3 Servicios *61*
4.5.4 Crítica *61*
4.5.5 Características interesantes *61*
4.6 GUIDE 61
4.6.1 Modelo de objetos *62*
4.6.2 Modelo de ejecución *62*
4.6.3 Invocación de objetos *62*
4.6.4 Migración de objetos *63*
4.6.5 Crítica *63*
4.6.6 Características interesantes *63*

4.7 CLOUDS....64
4.7.1 Abstracciones de Clouds....64
4.7.2 Distribución en Clouds....64
4.7.3 Espacio de objetos....65
4.7.4 Crítica....65
4.7.5 Características interesantes....65
4.8 SPRING....66
4.8.1 Abstracciones....66
4.8.2 Invocación de objetos....66
4.8.3 Nombrado....67
4.8.4 Seguridad....67
4.8.5 Crítica....67
4.8.6 Características interesantes....68
4.9 SOS....68
4.9.1 Modelo de objetos....68
4.9.2 Distribución de objetos....68
4.9.3 Migración de objetos....69
4.9.4 Modelo genérico....69
4.9.5 Diseño del sistema operativo....69
4.9.6 Crítica....70
4.9.7 Características interesantes....70
4.10 AMOEBA....70
4.10.1 Objetos de Amoeba....71
4.10.2 Capacidades....72
4.10.3 Nombrado de objetos....72
4.10.4 Crítica....72
4.10.5 Características interesantes....73
4.11 EMERALD....73
4.11.1 Objetos de Emerald....73
4.11.2 Actividad interna de los objetos....73
4.11.3 Comunicación y localización....73
4.11.4 Migración de objetos....74
4.11.5 Agrupación de objetos....74
4.11.6 Crítica....74
4.11.7 Características interesantes....75
4.12 COOLV2....75
4.12.1 Abstracciones base....75
4.12.2 Soporte genérico en tiempo de ejecución....75
4.12.3 Soportes específicos para lenguajes....76
4.12.4 Modelo de objetos....76
4.12.5 Invocación de objetos....76
4.12.6 Distribución....76
4.12.7 Crítica....77
4.12.8 Características interesantes....77
4.13 APERTOS....77
4.13.1 Estructuración mediante objetos y meta-objetos....77
4.13.2 Flexibilidad....78
4.13.3 Reflectividad....78
4.13.4 Jerarquía de reflectores....78
4.13.5 Invocación de objetos....79
4.13.6 Migración de objetos....79
4.13.7 Crítica....79
4.13.8 Características interesantes....79

CAPÍTULO 5 NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 81

5.1 INTRODUCCIÓN....81
5.2 NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS....81
5.3 CARACTERÍSTICAS DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS....82
5.3.1 Uniformidad conceptual en torno a la orientación a objetos....82
5.3.2 Transparencia....83

5.3.3 Heterogeneidad y portabilidad *83*
5.3.4 Seguridad *83*
5.3.5 Concurrencia *83*
5.3.6 Multilenguaje / Interoperabilidad *84*
5.3.7 Flexibilidad *84*
5.4 MODELO DE OBJETOS DEL SISTEMA INTEGRAL 85
5.5 RESUMEN 86
CAPÍTULO 6 ARQUITECTURA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS 87
6.1 INTRODUCCIÓN 87
6.2 ¿CÓMO OFRECER LA FUNCIONALIDAD DE LA DISTRIBUCIÓN DE OBJETOS? 87
6.2.1 Implementación ad-hoc en el núcleo básico de soporte de objetos *87*
6.2.2 Introducción de clases y objetos especiales que proporcionen la distribución *88*
6.2.3 Extensión no intrusiva del núcleo básico *88*
6.3 SISTEMA INTEGRAL OO = MAQUINA ABSTRACTA OO + SISTEMA OPERATIVO OO 89
6.3.1 Propiedades fundamentales de la arquitectura *89*
6.4 RESUMEN 91
CAPÍTULO 7 ARQUITECTURA DE REFERENCIA DE LA MÁQUINA ABSTRACTA 93
7.1 INTRODUCCIÓN 93
7.2 PROPIEDADES FUNDAMENTALES DE UNA MÁQUINA ABSTRACTA PARA UN SIOO 93
7.3 ESTRUCTURA DE REFERENCIA 94
7.4 JUEGO DE INSTRUCCIONES 95
7.4.1 Instrucciones declarativas *95*
7.4.2 Instrucciones de comportamiento *95*
7.5 SOPORTE BÁSICO AL MODELO DE OBJETOS 96
7.5.1 Modelo básico de objetos *96*
7.5.2 Polimorfismo *98*
7.5.3 Excepciones *98*
7.6 VENTAJAS DEL USO DE UNA MÁQUINA ABSTRACTA 98
7.6.1 Portabilidad y heterogeneidad *98*
7.6.2 Facilidad de comprensión *98*
7.6.3 Facilidad de desarrollo *98*
7.6.4 Compiladores de lenguajes *98*
7.6.5 Implementación de la máquina *99*
7.6.6 Buena plataforma de investigación *99*
7.7 MINIMIZACIÓN DEL PROBLEMA DEL RENDIMIENTO DE LAS MÁQUINAS ABSTRACTAS 99
7.7.1 Compromiso entre velocidad y conveniencia aceptado por los usuarios *100*
7.7.2 Mejoras en el rendimiento *100*
7.8 RESUMEN 100
CAPÍTULO 8 REFLECTIVIDAD PARA LA DISTRIBUCIÓN DE OBJETOS 101
8.1 INTRODUCCIÓN 101
8.2 REFLEXIÓN: MODIFICACIÓN DE LA ESTRUCTURA O EL COMPORTAMIENTO DE UN SISTEMA 101
8.3 SISTEMA BASE Y META-SISTEMA 102
8.3.1 Torre reflectiva *102*
8.4 ARQUITECTURAS REFLECTIVAS 103
8.4.1 Reflejo del meta-sistema o meta-modelo *103*
8.4.2 Actividades básicas en un sistema reflectivo: exposición y reflexión *105*
8.5 APLICACIÓN DE LA REFLECTIVIDAD EN SISTEMAS ORIENTADOS A OBJETOS 106
8.5.1 Representación del modelo del sistema como un conjunto de objetos *107*
8.6 META-INTERACCIÓN: TRANSFERENCIA DE CONTROL 108
8.6.1 Propiedades *108*
8.6.2 Clasificación de arquitecturas reflectivas atendiendo a la transferencia de control *109*
8.7 TIPOS DE REFLECTIVIDAD 109
8.7.1 Reflectividad estructural *109*
8.7.2 Reflectividad del comportamiento *110*
8.8 MODELOS DE REFLECTIVIDAD 111
8.8.1 Modelo meta-clase *112*
8.8.2 Modelo meta-objeto *114*

8.8.3 La reflectividad como exposición de la comunicación o exposición de los mensajes............ 116
8.9 META-INTERFACES Y MOP..117
8.9.1 Interfaz base versus interfaz meta.. 117
8.9.2 Protocolo de Meta-Objeto (MOP) 118
8.10 REFLECTIVIDAD Y SISTEMAS OPERATIVOS..119
8.10.1 Arquitectura reflectiva de un sistema operativo basado en meta-objetos............................. 119
8.10.2 Separación objeto/meta-objeto 119
8.11 RESUMEN..120

CAPÍTULO 9 EXTENSIÓN REFLECTIVA DE LA MÁQUINA ABSTRACTA PARA LA DISTRIBUCIÓN.. 121

9.1 INTRODUCCIÓN...121
9.2 ARQUITECTURA REFLECTIVA PROPUESTA...121
9.2.1 Torre reflectiva: dos niveles de objetos.. 121
9.2.2 Modelo de reflectividad 122
9.2.3 Descripción separada del nivel base y el meta-nivel 123
9.2.4 Utilización del paradigma de OO para describir el meta-nivel... 123
9.2.5 El nivel base.. 124
9.2.6 El meta-nivel.. 124
9.3 REFLECTIVIDAD ESTRUCTURAL 125
9.3.1 Exposición de la arquitectura interna de la máquina abstracta 126
9.3.2 Exposición de la implantación en tiempo de ejecución de los objetos.................................... 127
9.3.3 Exposición del meta-espacio de los objetos.. 128
9.3.4 La reflectividad estructural en la distribución de objetos... 129
9.4 REFLECTIVIDAD DEL COMPORTAMIENTO 129
9.4.1 Aspectos de la máquina que migran inicialmente al meta-nivel 130
9.4.2 La reflectividad del comportamiento en la distribución de objetos... 132
9.5 VENTAJAS DEL USO DE UNA ARQUITECTURA REFLECTIVA PARA INTRODUCIR LA DISTRIBUCIÓN EN EL SISTEMA INTEGRAL..139
9.5.1 Mantenimiento de la uniformidad conceptual... 139
9.5.2 Extensibilidad y adaptabilidad.. 139
9.5.3 Separación de asuntos o incumbencias (concerns) 139
9.5.4 Favorece el diseño de un modelo de objetos activo 140
9.5.5 Configurabilidad... 140
9.6 RESUMEN..140

CAPÍTULO 10 CUESTIONES INICIALES DE DISEÑO DE AGRA... 143

10.1 INTRODUCCIÓN..143
10.2 INTRODUCCIÓN DE LA DISTRIBUCIÓN EN EL SISTEMA INTEGRAL..144
10.3 CUESTIONES DE DISEÑO IMPUESTAS POR EL SISTEMA INTEGRAL...144
10.3.1 Uniformidad en torno a la orientación a objetos.. 145
10.3.2 Identidad de los objetos.. 145
10.4 CARACTERÍSTICAS DE LOS OBJETOS..146
10.4.1 Objetos activos autocontenidos.. 146
10.4.2 Granularidad de los objetos 149
10.4.3 Objetos compartidos... 150
10.4.4 Objetos de una copia.. 151
10.4.5 Objetos persistentes.. 151
10.4.6 Objetos móviles 152
10.4.7 Enlace dinámico 154
10.5 REPRESENTACIÓN INTERNA DE LOS OBJETOS...154
10.5.1 La clase sombra 156
10.6 ASPECTOS SEMÁNTICOS DE LA DISTRIBUCIÓN...157
10.6.1 ¿Debe ser transparente la distribución?.. 157
10.6.2 Cuándo mover un objeto 158
10.6.3 Semántica del paso de parámetros... 159
10.6.4 Cuestiones relativas a la concurrencia.. 161
10.7 RESUMEN...162

CAPÍTULO 11 GESTIÓN BÁSICA DE OBJETOS ... 163

11.1 INTRODUCCIÓN ... 163
11.2 NOMBRADO DE OBJETOS ... 164
11.2.1 Identificadores y referencias ... 164
11.2.2 Nombres simbólicos ... 165
11.2.3 Políticas de nombrado ... 166
11.3 LOCALIZACIÓN DE OBJETOS ... 166
11.3.1 Semánticas de localización ... 167
11.3.2 Localización de objetos en el sistema integral ... 168
11.3.3 Opciones de localización ... 171
11.3.4 Localización fuerte ... 172
11.3.5 Otras políticas ... 173
11.4 CREACIÓN Y ELIMINACIÓN DE OBJETOS ... 174
11.4.1 Creación de un objeto ... 175
11.4.2 Eliminación de un objeto ... 178
11.5 COMPROBACIÓN DE TIPOS ... 180
11.5.1 Optimizaciones para la comprobación de tipos ... 181
11.6 INTERACCIÓN CON LA RED DE COMUNICACIONES ... 181
11.7 RESUMEN ... 182

CAPÍTULO 12 INVOCACIÓN REMOTA Y MIGRACIÓN DE OBJETOS ... 185

12.1 INTRODUCCIÓN ... 185
12.2 INVOCACIÓN DE MÉTODOS ... 185
12.2.1 Invocación síncrona ... 186
12.2.2 Invocación asíncrona ... 195
12.3 PROPAGACIÓN DE EXCEPCIONES ... 196
12.4 MIGRACIÓN DE OBJETOS ... 197
12.4.1 Problemas que presenta la migración de objetos ... 197
12.4.2 Decisiones de diseño ... 197
12.4.3 Migración de objetos en el sistema integral ... 199
12.4.4 Cómo mover un objeto ... 199
12.4.5 Cuándo, qué y adónde migrar objetos ... 207
12.5 GESTIÓN DE FALLOS DE NODOS ... 210
12.6 RESUMEN ... 211

CAPÍTULO 13 ASPECTOS ADICIONALES RELACIONADOS CON LA DISTRIBUCIÓN ... 213

13.1 INTRODUCCIÓN ... 213
13.2 SERVICIO DE NOMBRADO ... 213
13.2.1 Servicio de nombrado para el sistema integral ... 214
13.3 TRANSACCIONES ... 216
13.3.1 Transacciones software ... 217
13.3.2 Introducción de transacciones en el sistema integral ... 219
13.4 REPLICACIÓN ... 220
13.4.1 Introducción de objetos replicados en el sistema integral ... 222
13.5 RECOLECCIÓN DE BASURA DISTRIBUIDA ... 223
13.5.1 Algoritmos de recolección de basura ... 224
13.5.2 Recolección de basura en el sistema integral ... 225
13.6 RESUMEN ... 227

CAPÍTULO 14 RELACIÓN DEL SUBSISTEMA DE DISTRIBUCIÓN CON OTROS SUBSISTEMAS ... 229

14.1 INTRODUCCIÓN ... 229
14.2 SUBSISTEMA DE PERSISTENCIA ... 229
14.2.1 Elementos básicos de diseño del sistema de persistencia ... 229
14.2.2 Implementación de la persistencia en el sistema integral ... 231
14.2.3 Relación con la distribución ... 233
14.3 SUBSISTEMA DE PROTECCIÓN ... 234
14.3.1 Requisitos de protección ... 234
14.3.2 Modelo de protección ... 235

14.3.3 Implantación en la máquina abstracta ... *237*
14.3.4 Modo de operación del mecanismo ... *238*
14.3.5 Relación con la distribución ... *239*
14.4 RESUMEN ... 241

CAPÍTULO 15 EL SISTEMA INTEGRAL ORIENTADO A OBJETOS OVIEDO3 ... 243

15.1 INTRODUCCIÓN ... 243
15.2 LA MÁQUINA ABSTRACTA CARBAYONIA ... 244
15.2.1 Estructura ... *244*
15.2.2 El lenguaje Carbayón: juego de instrucciones ... *246*
15.2.3 Características de las clases Carbayonia ... *247*
15.2.4 Jerarquía de clases básicas ... *248*
15.2.5 Clase básica objeto ... *249*
15.3 EL SISTEMA OPERATIVO SO4 ... 251
15.4 RESUMEN ... 252

CAPÍTULO 16 IMPLEMENTACIÓN DE UN PROTOTIPO DE DISTRIBUCIÓN DE OBJETOS SOBRE EL SISTEMA INTEGRAL OVIEDO3 ... 253

16.1 INTRODUCCIÓN ... 253
16.2 PRESENTACIÓN DE LA MÁQUINA ABSTRACTA CARBAYONIA ... 253
16.2.1 Diseño de la máquina como conjunto de objetos ... *253*
16.3 DESCRIPCIÓN DE LA JERARQUÍA DE CLASES ... 254
16.4 PROTOTIPO DE DISTRIBUCIÓN NO REFLECTIVO ... 255
16.4.1 Problemas presentados para la distribución por la máquina abstracta original ... *255*
16.4.2 Aspectos iniciales de diseño ... *258*
16.4.3 Invocación remota ... *258*
16.4.4 Propagación de excepciones ... *259*
16.5 PROTOTIPO DE DISTRIBUCIÓN REFLECTIVO ... 260
16.5.1 Clases primitivas para la reflectividad estructural ... *261*
16.5.2 Representación interna de la reflectividad estructural ... *268*
16.5.3 Clases primitivas para la reflectividad del comportamiento: implantación del sistema operativo ... *271*
16.5.4 Representación interna de la reflectividad del comportamiento ... *277*
16.5.5 Descripción dinámica del prototipo ... *278*

CAPÍTULO 17 TRABAJO RELACIONADO ... 285

17.1 INTRODUCCIÓN ... 285
17.2 META-JAVA ... 285
17.2.1 Guías de diseño ... *285*
17.2.2 Modelo computacional de MetaJava ... *285*
17.2.3 Asociación de meta-objetos con las entidades del nivel base ... *287*
17.2.4 Estructura ... *287*
17.2.5 Reflectividad del comportamiento ... *288*
17.2.6 Implantación de la relación objetos base – meta-objetos: clases sombra ... *288*
17.2.7 Transferencia del nivel base al meta-nivel: eventos ... *290*
17.2.8 Invocación de métodos remotos ... *290*
17.2.9 Implantación ... *291*
17.3 AGENTES MÓVILES ... 292
17.3.1 Definición de agente ... *292*
17.3.2 Movilidad ... *293*
17.3.3 El paradigma de agentes móviles ... *293*
17.3.4 Conceptos de agentes móviles ... *294*
17.3.5 Tecnología de agentes móviles ... *296*
17.3.6 Particularidades de los agentes móviles para la construcción de sistemas distribuidos ... *298*
17.3.7 Aplicaciones de los agentes móviles ... *298*

CAPÍTULO 18 CONCLUSIONES ... 301

18.1 INTRODUCCIÓN ... 301
18.2 RESULTADOS DESTACABLES ... 302
18.2.1 Arquitectura reflectiva del sistema integral ... *302*

18.2.2 Introducción reflectiva de la distribución *302*
18.2.3 Transparencia *303*
18.2.4 Control de la distribución para ciertas aplicaciones *303*
18.2.5 Uniformidad *303*
18.2.6 Adaptabilidad *304*
18.2.7 Flexibilidad *304*
18.3 TRABAJO Y LÍNEAS DE INVESTIGACIÓN FUTURAS 304
18.3.1 Interoperabilidad con otros sistemas de soporte de objetos *304*
18.3.2 Construcción de compiladores de lenguajes *305*
18.3.3 Implementación de servicios *305*
18.3.4 Introducción de otros servicios relacionados con la distribución *306*
18.3.5 Desarrollo completo de un entorno de usuario *306*

REFERENCIAS 307

CAPÍTULO 1 INTRODUCCIÓN, OBJETIVOS Y ORGANIZACIÓN DE LA TESIS

Con la utilización generalizada de ordenadores personales, estaciones de trabajo y redes de área local, los sistemas distribuidos están siendo cada vez más comunes. Este tipo de sistemas es inherentemente más complejo que los no distribuidos y la programación de aplicaciones que se aprovechan de dicha distribución sufre también de dicha complejidad. Se han propuesto distintos sistemas operativos y lenguajes de programación para reducir la complejidad mencionada.

Las aplicaciones distribuidas necesitan, de manera habitual la compartición de datos entre entidades remotas. Tal compartición se consigue accediendo a datos remotos compartidos o copiando los datos compartidos entre las entidades que cooperan. Los sistemas y lenguajes existentes prohiben, en general, la compartición distribuida de los datos o proporcionan dos niveles de soporte, uno para los datos contenidos totalmente en una máquina (o incluso proceso) y otro para los datos que son compartidos superando las barreras entre máquinas diferentes. Muchos sistemas proporcionan una forma limitada de compartición permitiendo la compartición-por-copia de pequeñas cantidades de datos, por ejemplo, incluyendo los datos en mensajes o como parámetros en llamadas a procedimientos remotos. Este tipo de copia se denomina de grano fino, porque los datos copiados pueden ser arbitrariamente pequeños.

Por otro lado, la orientación a objetos se presenta como un paradigma estandarizado de hecho que es utilizado en el desarrollo de aplicaciones por las ventajas que tiene. Entre estas ventajas está el uso de un único paradigma en las diferentes fases del ciclo de vida del software: análisis de requerimientos OO, análisis OO, diseño OO e implementación con lenguajes de programación OO. Los sistemas resultantes son más fáciles de entender y de desarrollar, pues se reducen los saltos semánticos al aplicar los mismos conceptos en todas las fases.

Esta Tesis describe la forma en que se pueden utilizar las tecnologías orientadas a objetos para la construcción de un sistema operativo distribuido que se convertirá, en definitiva, en un sistema operativo distribuido orientado a objetos. Para conocer los objetivos que se desean cubrir con este trabajo, es necesario comenzar describiendo los problemas iniciales que surgen de la combinación de sistemas distribuidos y orientación a objetos en su utilización habitual.

1.1 Problemas de la utilización convencional de las tecnologías orientadas a objetos para la construcción de sistemas distribuidos

1.1.1 El problema de la desadaptación de impedancias o salto semántico

A pesar del éxito de las tecnologías orientadas a objetos (TOO) en el ámbito del desarrollo de aplicaciones, la adopción del paradigma OO no se ha hecho de una manera integral en todos los elementos que componen un sistema de computación. Existe un grave problema de desadaptación de impedancias, o salto semántico entre los diferentes

elementos del sistema. Este problema se produce en el momento en que hay que realizar un cambio o adaptación de paradigma cuando un elemento (por ejemplo una aplicación OO) debe interactuar con otro (por ejemplo el sistema operativo).

1.1.1.1 Abstracciones no adecuadas de los sistemas operativos

Por una parte, el hardware convencional sobre el que deben funcionar las TOO sigue aún basado en versiones evolucionadas de la arquitectura Von Neumann. Los sistemas operativos ofrecen abstracciones basadas en este tipo de hardware, más adecuadas para el paradigma procedimental estructurado. Así los contextos de ejecución y espacios de direcciones de los procesadores convencionales tienen sus abstracciones correspondientes en el sistema operativo: procesos y memoria virtual, hilos de ejecución y mecanismos de comunicación entre procesos (IPC, Inter-Process Communication).

Las aplicaciones OO se estructuran básicamente como un conjunto de objetos que colaboran entre sí mediante la invocación de métodos. La distancia semántica entre esta estructura y las abstracciones que ofrecen los sistemas operativos es muy grande.

Por ejemplo, los objetos de los lenguajes en que se desarrollan las aplicaciones suelen ser de grano fino (tamaño pequeño). Sin embargo, el elemento más pequeño que manejan los sistemas operativos es el de un proceso asociado a un espacio de direcciones, de un tamaño mucho mayor. Esto obliga a que sea el compilador del lenguaje el que estructure los objetos internamente dentro de un espacio de direcciones. Pero por ejemplo, al desarrollar aplicaciones distribuidas esto ya no funciona puesto que el compilador desconoce la existencia de otros objetos fuera de un espacio de direcciones. Por otro lado, los modelos de concurrencia de objetos suelen disponer de actividades ligeras u objetos activos que deben hacerse corresponder de manera forzada con la noción más gruesa de proceso.

Comunicación de alto nivel entre objetos situados en diferentes espacios de direcciones

Un ejemplo de este problema es el mencionado de la comunicación de alto nivel entre objetos de diferentes espacios de direcciones. Se plantea cuando un objeto cliente tiene que invocar un método que ofrece un objeto servidor y los dos objetos no están dentro del mismo espacio de direcciones (métodos remotos). Por ejemplo se da este caso cuando los dos objetos están en máquinas diferentes. En este caso el compilador no tiene ningún mecanismo que permita realizar la invocación de métodos en espacios de direcciones diferentes, y se debe recurrir a los mecanismos que ofrezca el sistema operativo. Sin embargo, el mecanismo de comunicación de los sistemas operativos no se adapta al paradigma de la OO, ya que están orientados a comunicar procesos. Un ejemplo de estos mecanismos son las tuberías o pipes del SO Unix. El programador se ve obligado a abandonar el paradigma OO y encajar de manera antinatural el mecanismo de comunicación OO (invocación de métodos) sobre un mecanismo totalmente distinto pensado para otra finalidad.

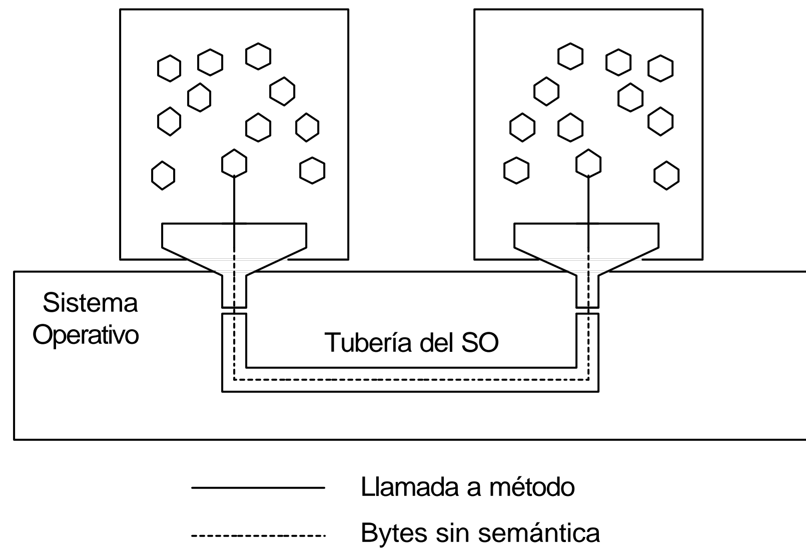


Figura 1.1. Comunicación entre objetos mediante mecanismos de bajo nivel del sistema operativo.

La implementación de la OO sobre estos sistemas que no tienen soporte explícito para objetos es muy complicada, como demuestra la implementación del lenguaje Eiffel concurrente [KB90]. Para solucionar los problemas anteriores se acaba recurriendo a la interposición de capas adicionales de software que adapten la gran diferencia existente entre el paradigma OO y los sistemas actuales. Ejemplos de estas capas de software son CORBA y COM, ya descritos en el capítulo anterior.

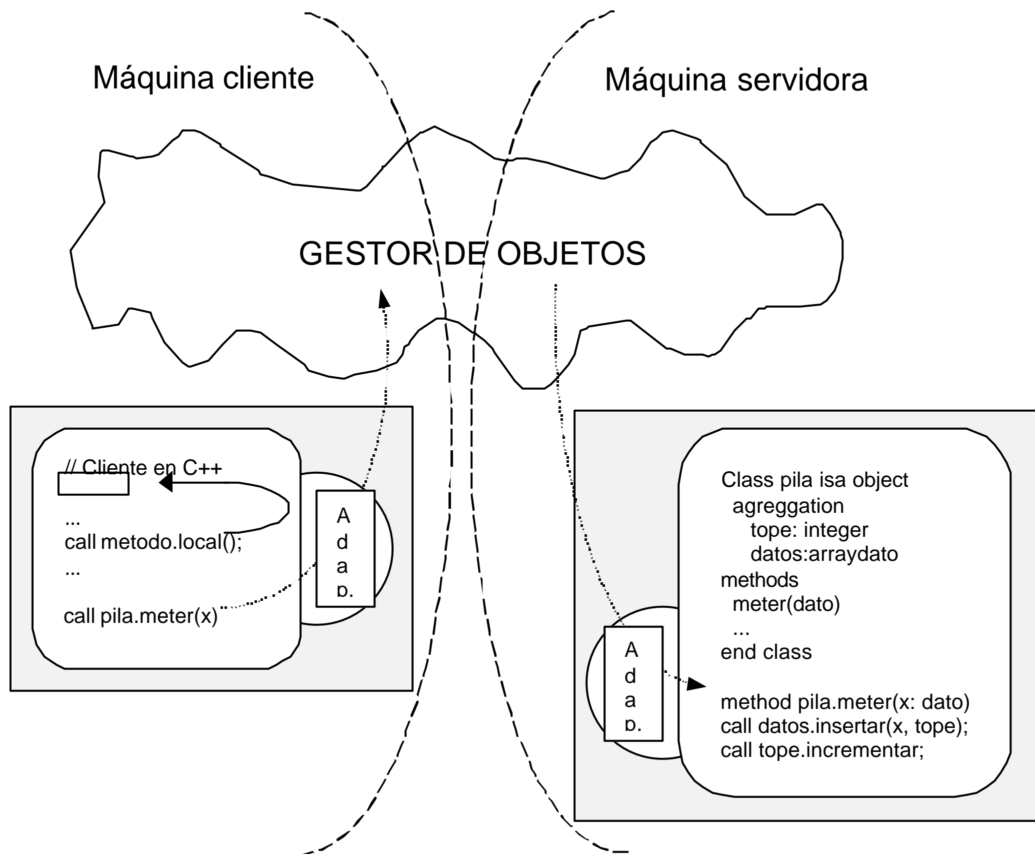


Figura 1.2. Comunicación de alto nivel mediante un gestor de objetos

1.1.1.2 Desadaptación de interfaces

Otro problema se produce cuando las aplicaciones OO necesitan utilizar los servicios del sistema operativo. La interfaz de utilización de estos servicios está enfocado al paradigma procedimental, normalmente en forma de una llamada al sistema (llamada a procedimiento). Además, como ya se mencionó anteriormente, muchos de estos servicios (por ejemplo la comunicación entre procesos) no sigue el paradigma OO.

El resultado es que el programador/usuario del sistema se ve obligado a utilizar dos paradigmas diferentes en el desarrollo de las aplicaciones. Uno para la parte fundamental de la aplicación, orientado a objetos, y otro totalmente diferente para la interacción con el sistema operativo.

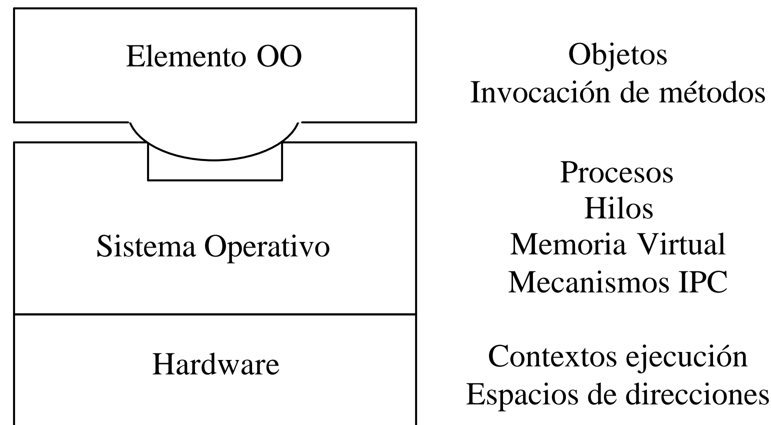


Figura 1.3. Desadaptación entre las aplicaciones OO y la interfaz del sistema operativo.

Por ejemplo, en el caso anterior, para la utilización de una tubería en Unix no se puede utilizar la OO. Es necesario utilizar llamadas de procedimiento a la interfaz procedimental del SO, las llamadas al sistema. O bien para utilizar los ficheros que proporciona el sistema operativo, etc.

Esta dualidad de paradigmas produce una disminución de la productividad de los programadores. Cuantos más sean los conceptos diferentes que deba conocer el programador, peor será la comprensión y el dominio de los mismos. La necesidad de aplicar otro paradigma diferente a la orientación a objetos hace que no se saque todo el partido posible a la misma.

También se pueden utilizar capas de adaptación, por ejemplo encapsulando la interfaz del sistema operativo mediante una librería de clases. Así las llamadas al sistema operativo se hacen indirectamente a través de objetos de la librería de clase, de manera OO. La librería, a su vez, se encarga de llamar al sistema operativo.

Un ejemplo de estas capas de adaptación es la librería MFC (Microsoft Foundation Classes [Mic97a, Mic97b]), que entre otras cosas, encapsula ciertos servicios del sistema Windows en un marco de aplicación.

1.1.2 El problema de la interoperabilidad entre modelos de objetos

Incluso aunque diferentes elementos del sistema usen el paradigma de la orientación a objetos, puede haber problemas de desadaptación entre ellos. Es común la existencia de diferentes lenguajes de programación OO, bases de datos OO, interfaces gráficas OO, etc. Sin embargo, el modelo de objetos que utiliza cada uno de ellos suele ser diferente. Aunque sean OO, las propiedades de los objetos de cada sistema pueden diferir, por ejemplo un modelo puede tener constructores y otros no, etc.

Por ejemplo, una aplicación desarrollada usando el lenguaje C++, con el modelo de objetos C++ no tiene ningún problema de comunicación con sus propios objetos. Cuando se pretende usar objetos de otro lenguaje de programación, o interactuar con objetos que no están en el mismo proceso, o bien con una base de datos orientada a objetos aparece un problema de interoperabilidad. Este es debido a que el modelo de objetos del C++ no tiene por qué ser totalmente compatible con el de los otros elementos.

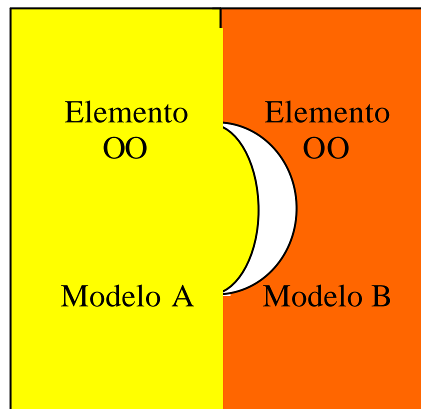


Figura 1.4. Desadaptación entre modelos de objetos diferentes.

De nuevo se recurre a la introducción de capas de software de adaptación para solucionar el problema. CORBA también es un ejemplo de este tipo de software.

1.1.3 Problemas de las capas de adaptación sobre sistemas tradicionales

Para salvar los problemas que presenta el paradigma de la orientación a objetos, se utilizan soluciones basadas en la adición de una serie de capas de software de adaptación a un sistema operativo tradicional. Esto provoca una serie de inconvenientes:

Disminución del rendimiento global del sistema, a causa de la sobrecarga debida a la necesidad de atravesar todas estas capas software para adaptar los paradigmas. Además, para la propia implementación de estas capas que ofrecen soporte de objetos se necesitan también los servicios del sistema operativo, con lo que nos encontramos con un nuevo salto entre paradigmas

Falta de uniformidad y transparencia. En la práctica, estas soluciones no son todo lo transparente que deberían de ser de cara al usuario. Por ejemplo, la escritura de objetos que vayan a funcionar como servidores suele realizarse de manera diferente del resto de los objetos, como en el caso de CORBA. Esto no se corresponde con la filosofía de uso y de programación OO, que es más general que este paradigma cliente/servidor que obliga a clasificar los objetos en clientes o servidores a priori.

Pérdida de portabilidad y flexibilidad. La falta de uniformidad produce una pérdida de portabilidad y flexibilidad. El programador se ve forzado a utilizar ciertas convenciones, formas especiales de programar, etc. que impone el uso de la propia capa que reduce la portabilidad de los objetos. Por ejemplo, en el caso de utilizar la librería MFC para utilizar las funciones de Windows se impone una determinada manera de construir los programas, etc. que hace que el código quede ligado indisolublemente a esta librería. Otro ejemplo es el caso de un programa que tenga que utilizar objetos CORBA, su programación es totalmente distinta que si se va a utilizar objetos COM. Además, estos programas sólo podrán funcionar en máquinas a las que también se haya portado los sistemas de adaptación que se utilicen.

Aumento de la complejidad del sistema. La adición de estas capas introduce más complejidad en los sistemas. Por un lado aumentan los problemas de integración entre las diferentes capas, errores en la misma, posibilidades de fallos, etc. Por otro lado, los sistemas se hacen más difíciles de comprender al intervenir tantos elementos y tan dispares.

Soluciones parciales. En general, estas capas sólo ofrecen soluciones parciales a alguno de los problemas presentados. Por ejemplo, para la invocación de métodos remotos pueden existir soluciones, sin embargo no solucionan la utilización OO de recursos del sistema operativo, etc. Esto obliga a combinar diferentes capas para solucionar diferentes aspectos. CORBA es un ejemplo que soluciona la interoperabilidad entre objetos, pero colocándolos en espacios de direcciones diferentes. En el desarrollo intra-programa no puede usarse CORBA, con lo que se pierde la interoperabilidad entre objetos de diferentes lenguajes.

Pérdida de productividad. Todo lo anterior lleva a una pérdida de productividad. La programación se hace a veces engorrosa e incluso compleja. Se distrae al programador con detalles técnicos de la capa (o capas) de adaptación y se le impide concentrarse en la solución de los problemas.

En general, el problema es debido a que los sistemas operativos no soportan el concepto de objeto. Por tanto este concepto es proporcionado internamente por los compiladores de cada lenguaje, estructurándolo sobre las abstracciones que proporciona el sistema operativo. El desconocimiento de la existencia de objetos por parte del SO provoca una serie de problemas que no tienen una solución satisfactoria con la adición de "parches" o capas de software adicional a los sistemas existentes.

1.2 Objetivos

En esta Tesis se trata de desarrollar el núcleo básico de la distribución de objetos para un Sistema Operativo Distribuido Orientado a Objetos (SODOO), en el que se integren todos los beneficios aportados por los sistemas distribuidos y las tecnologías orientadas a objetos y tratando de minimizar sus inconvenientes.

A continuación se describen brevemente los objetivos a cubrir, que son consecuencia de los problemas mostrados anteriormente dentro de este capítulo y de los que posteriormente se irán identificando en las soluciones existentes que se detallan en los capítulos 2, 3 y 4 de este trabajo.

1.2.1 El objeto como abstracción única

La utilización del objeto como única abstracción es el objetivo inicial de este trabajo. Como ya se ha comentado anteriormente, en muchos sistemas se produce un problema de desadaptación de impedancias por la utilización de diferentes paradigmas para la resolución de un problema. En este caso, se trata de construir un sistema operativo distribuido orientado a objetos que proporcione sus servicios a las aplicaciones.

No se quiere realizar, por tanto, una aplicación parcial de las tecnologías de objetos para la construcción del sistema operativo, sino que se quiere llevar hasta sus últimas consecuencias. De esta manera, el sistema operativo estará construido como un conjunto de objetos, ofrecerá sus servicios en forma de objetos y las aplicaciones de usuario también estarán formadas por un conjunto de objetos. Desaparece, entonces, el concepto de proceso como abstracción principal del sistema operativo que describe un espacio de direcciones en el que se almacenan las instrucciones y los datos que manejan estas, y aparece el objeto, como abstracción única del sistema, y que encapsulará datos y computación.

1.2.2 Modelo de objetos uniforme

En los sistemas distribuidos, los computadores están separados físicamente por la red y los programadores tienen que realizar dos tipos de operaciones: locales y remotas. Los sistemas que soportan ambos modelos de computación tienen un mecanismo separado para cada modelo, de tal forma que el programador realiza un conjunto de acciones para utilizar objetos locales y un conjunto de acciones distinto para acceder a objetos remotos. Es evidente que la presencia de ambos modelos complica notablemente la programación.

En esta Tesis se persigue la utilización de un modelo de objetos uniforme, tanto para los objetos locales como para los remotos. Así mismo, todos los servicios proporcionados por el sistema operativo se realizarán en forma de objetos, que siguen el mismo modelo que los objetos de usuario. Por tanto, los objetos se utilizarán de la misma forma, con independencia de su ubicación y de su naturaleza (del sistema o de usuario).

1.2.3 Transparencia

La transparencia es un objetivo esencial en la construcción de cualquier tipo de sistema distribuido. Es importante que el programador no tenga que estar al tanto de la naturaleza distribuida del sistema y que aun así pueda beneficiarse de la misma. La transparencia está muy relacionada con la uniformidad del modelo de objetos en el sentido de que los objetos locales y remotos se tienen que utilizar de la misma forma, sin necesidad de intervención alguna (e incluso conocimiento) por parte del programador. De todas formas, y para aquellos programadores que deseen obtener un beneficio particular del conocimiento de la arquitectura distribuida del sistema para la construcción de sus aplicaciones, el sistema operativo proporcionará ciertos mecanismos para el control de la distribución.

1.2.4 Migración de grano fino

La migración en los sistemas operativos distribuidos se convierte en una potente herramienta para conseguir una mejora del rendimiento global de los mismos. Es posible aprovechar aquellas máquinas con menor carga computacional para que ejecuten trabajos que de otra manera se ejecutarían más lentamente en máquinas más cargadas. Generalmente, la migración ha sido considerada para los procesos, donde las entidades que se mueven son espacios de direcciones completos. Entidades más pequeñas, como por ejemplo registros, no pueden migrar independientemente.

En un sistema operativo distribuido orientado a objetos, en el que la abstracción única es el objeto, el objeto se convierte en la unidad de migración. Un objeto, de cualquier tamaño, puede migrar de una máquina a otra, obteniéndose una migración de grano fino. Además, dado que los objetos contienen su propia computación, la migración de grano fino incluye tanto la migración de datos como la tradicional migración de procesos.

1.2.5 Flexibilidad, extensibilidad, adaptabilidad

En los sistemas operativos tradicionales, el conjunto de recursos y servicios que proporcionan a las aplicaciones están restringidos por las llamadas al sistema. En algunos de ellos, incluso, dichos recursos y servicios están implantados rígidamente en su núcleo, de manera que no pueden ser modificados para adaptarlos a un entorno de ejecución concreto.

El sistema operativo distribuido orientado a objetos a diseñar tiene que superar todos los inconvenientes mencionados. En primer lugar, tiene que ser flexible, de tal manera que se diseñe un núcleo básico de distribución de objetos que permita construir sobre él mecanismos de más alto nivel y diferentes políticas. En segundo lugar, tiene que ser extensible, en el sentido que sea sencilla y no traumática la incorporación de nuevos servicios en el sistema. Finalmente, debe ser adaptable, de tal forma que se puedan escoger aquellas facilidades que el sistema operativo vaya a ofrecer para un entorno determinado, y sin que suponga un excesivo trastorno debido a las facilidades que no se ofrezcan.

1.2.6 Independencia entre mecanismos

La independencia entre mecanismos es fundamental para maximizar la adaptabilidad del sistema. Los mecanismos utilizados por las distintas facilidades del sistema operativo deben ser lo más interindependientes posibles, de tal forma que su presencia o ausencia no afecte al resto. Por ejemplo, los mecanismos de migración de objetos e invocación remota deben diseñarse de tal forma que una invocación pueda tener lugar con independencia de que el objeto esté en uno u otro nodo o que incluso esté migrando en ese momento. De la misma forma, un objeto debe ser invocado sin importar que esté ubicado en almacenamiento principal o en almacenamiento secundario, de una manera similar a lo que ocurre con la memoria virtual.

1.2.7 Independencia de mecanismos y políticas

Tal y como ocurre entre los mecanismos, es deseable conseguir la máxima independencia entre los mecanismos y las políticas que siguen estos. Dicha independencia permite mejorar también la adaptabilidad del sistema, de tal forma que para un entorno de ejecución concreto se puedan escoger las políticas más adecuadas para cada uno de los mecanismos del sistema operativo.

1.3 Proyecto de investigación en el que se enmarca la Tesis

Esta tesis se desarrolla dentro del Proyecto de Investigación Oviedo3, a cargo del Grupo de Investigación de Tecnologías Orientadas a Objetos, en el Departamento de Informática de la Universidad de Oviedo. En él se trata de diseñar un entorno de computación que utilice las tecnologías orientadas a objetos en toda su extensión, eliminando las desadaptaciones de impedancias producidas por los habituales cambios de paradigma. La gestión distribuida de los objetos del sistema constituye uno de sus aspectos principales y es el motivo de esta investigación.

1.4 Organización de los contenidos

Los contenidos de esta Tesis se agrupan en cuatro secciones: antecedentes, solución, prototipo y conclusiones. Cada una de las partes se constituye en una fase de trabajo encaminada a la solución del problema, desde la presentación del mismo hasta la construcción de un prototipo que incluye los mecanismos necesarios para su solución.

1.4.1 Antecedentes

La sección de antecedentes comprende los capítulos 2, 3 y 4. En ellos se describe la situación actual de la construcción de sistemas distribuidos orientados a objetos y cómo está siendo realizada de tal manera que no se cubren los objetivos ya mencionados.

El capítulo 2 sirve como introducción al mundo de los sistemas distribuidos, describiendo el conjunto de propiedades que los caracterizan y las ventajas e inconvenientes que presentan su diseño y utilización. También se presenta una definición informal de sistema operativo distribuido.

El capítulo 3 trata de la utilización de las tecnologías orientadas a objetos para la construcción de sistemas distribuidos. Se estudian los diferentes modelos de construcción y el alcance que puede llegar a tener la aplicación de dichas tecnologías. Finalmente, y en base a todo lo estudiado, se proporciona una definición de sistema operativo orientado a objetos y distribuido que se utilizará como base para delimitar el que posteriormente se va a diseñar.

El capítulo 4 está dedicado a la descripción de los sistemas distribuidos orientados a objetos más relevantes. Se incluyen en el capítulo tanto sistemas de soporte de objetos como CORBA, RMI y DCOM, que no son sistemas operativos, como verdaderos sistemas operativos, con el fin de tener una perspectiva más amplia.

1.4.2 Solución

La sección dedicada a la solución es la más amplia de esta Tesis, y comprende desde el capítulo 5 hasta el 14. En ella se propone la construcción de un Sistema Integral Orientado a Objetos que proporcione el soporte de objetos en sus capas más básicas como única abstracción del sistema. El sistema integral se compone de una máquina abstracta y un sistema operativo, ambos orientados a objetos. Con el fin de conseguir los objetivos marcados en la Tesis, será necesario dotar a la máquina abstracta de una arquitectura reflectiva. De esta forma, los diferentes mecanismos que componen la distribución de objetos se podrán introducir de una manera sencilla y no intrusiva para otros elementos.

El capítulo 5 sirve de puente entre la sección de antecedentes y la sección de la solución, identificando los problemas comunes de los sistemas distribuidos estudiados en el capítulo 4 y cómo un Sistema Integral Orientado a Objetos viene a solucionarlos en buena manera. El modelo de objetos del sistema integral seguirá las líneas básicas descritas en el capítulo 3.

El capítulo 6 afronta la construcción del sistema integral. Se discuten diferentes alternativas, resultando elegida la que define el sistema integral como una combinación de máquina abstracta y sistema operativo orientados a objetos. Queda aún por determinar de que manera van a colaborar ambos.

El capítulo 7 muestra una arquitectura de referencia para la máquina abstracta orientada a objetos, describiéndose para qué características del modelo de objetos del sistema integral proporciona soporte, el juego de instrucciones y las relaciones entre objetos que va a mantener.

El capítulo 8 está dedicado a la reflectividad, como herramienta a utilizar para la colaboración entre máquina abstracta y sistema operativo. Se trata de un capítulo de generalidades, y su aplicación será mostrada en capítulos posteriores.

El capítulo 9 proporciona el primer acercamiento a la utilización de la reflectividad en el sistema integral. Se describe la arquitectura reflectiva concreta que va a presentar la máquina abstracta y cuál va a ser la forma en que la reflectividad permita la introducción de la distribución en el sistema.

El capítulo 10 está dedicado a concretar ciertos aspectos de diseño del sistema de distribución. En el capítulo 3 se habían descrito diferentes alternativas para cada una de

las características que podría tener un objeto distribuido. En este momento se decide qué alternativas se han escogido y por qué.

El capítulo 11 trata las operaciones básicas de gestión de objetos que se ven afectadas por la introducción de la distribución. Se muestra de qué manera tienen que ser realizadas las operaciones de localización, nombrado, creación, eliminación y comprobación de tipos en el nuevo entorno distribuido. La interacción con la red de comunicaciones es un aspecto nuevo, que no tenía razón de ser en un entorno no-distribuido, y que también será tratado.

En el capítulo 12 se tratan los mecanismos centrales del sistema de distribución de objetos: la invocación remota y la migración. Se muestra cómo la reflectividad de las máquinas abstractas es fundamental para la implantación de estos mecanismos. La parte más importante del capítulo está dedicada a una descripción concienzuda de los diferentes pasos que permiten realizar cada una de las dos operaciones.

El capítulo 13 describe otros aspectos fuertemente relacionados con la distribución, pero que no se consideran integrantes del núcleo de la misma. Se trata de los servicios de nombrado, transacciones y replicación y la recolección de basura.

El capítulo 14 muestra de qué forma se relaciona el subsistema de distribución diseñado para el sistema integral con otros subsistemas diseñados en otros trabajos. La persistencia y la seguridad son fundamentales en los sistemas operativos orientados a objetos modernos y es necesario estudiar de qué manera se integran con lo diseñado en esta Tesis.

1.4.3 Prototipo

La sección dedicada al prototipo comprende los capítulos 15 y 16, y trata de mostrar una implementación real de lo descrito. Se parte de un prototipo inicial para el sistema integral, que incluye un sistema de distribución excesivamente rígido, con el fin de detectar los problemas principales que puede solucionar la solución elegida, que ya incluye una máquina abstracta reflectiva como la propuesta.

El capítulo 15 describe el sistema integral orientado a objetos Oviedo3, que se constituye en un objetivo de investigación de gran envergadura que incluye la presente Tesis y otra serie de ellas, dedicadas a diferentes aspectos del mismo como la persistencia, seguridad, lenguajes de programación, bases de datos, etc.

El capítulo 16 presenta varios prototipos del sistema de distribución realizados para el sistema integral. Se parte de un prototipo de la máquina abstracta a la que se dota de distribución sin hacer uso de la reflectividad. Dado que el resultado obtenido es excesivamente rígido, se realiza un segundo prototipo que soluciona los problemas planteados por el primero a través de la utilización de la reflectividad como herramienta arquitectónica.

1.4.4 Conclusiones

La última sección de esta Tesis, dedicada a las conclusiones, está formada por los capítulos 17 y 18.

El capítulo 17 no trata exactamente de las conclusiones, sino de los trabajos contemporáneos al descrito en este documento y que están relacionados con los temas tratados de una manera importante.

El capítulo 18 está dedicado a las conclusiones del trabajo y las líneas de investigación que quedan abiertas a partir del punto en que finaliza la actual.

CAPÍTULO 2 CONCEPTOS DE SISTEMAS DISTRIBUIDOS

2.1 Introducción

La utilidad de un sistema informático, el alcance y la calidad de los servicios que proporciona y su facilidad de uso dependen fuertemente de su sistema operativo [Gos91]. El **sistema operativo** puede ser definido como aquella parte del sistema informático que da vida al hardware. El desarrollo de los sistemas operativos va siempre detrás del desarrollo del hardware, pero permiten mejorar el rendimiento de este, y, en el peor de los casos, ocultarán todas sus particularidades.

Hasta hace bien poco tiempo los sistemas operativos se construían para sistemas informáticos centralizados. Existe una gran cantidad de literatura sobre la teoría del diseño y construcción de los mismos [SG98, Sta98, TW97].

En la actualidad nos encontramos un paso más allá en el desarrollo de sistemas informáticos. Se está investigando, por un lado, para incrementar la capacidad de procesamiento de computadores aislados, utilizando arquitecturas multiprocesador y, por otro lado, para unir la capacidad de procesamiento de computadores independientes dispersos geográficamente. Nos encontramos, en este último caso, con el **desarrollo de los sistemas distribuidos**.

A la hora de construir un sistema distribuido, hay que pararse a estudiar, entre otras cosas, su alcance, es decir, si el sistema a construir se va a limitar a aprovechar la capacidad de procesamiento para resolver un problema concreto o si, por el contrario, se desea construir una plataforma de distribución que facilite posteriormente la construcción de aplicaciones que aprovechen, de una manera más sencilla, dicha capacidad de procesamiento. En el primer caso se habla de la construcción de **aplicaciones distribuidas**. En el segundo caso se habla de **sistemas operativos distribuidos**.

Tal como ocurría en los sistemas operativos centralizados, el objetivo principal de un sistema operativo distribuido será **proporcionar un conjunto de servicios** (los convencionales, de sistemas operativos tradicionales y otros nuevos, propios de los sistemas operativos distribuidos) a los programas de usuario de una manera sencilla, con las ventajas y los inconvenientes que lleva consigo el trabajar sobre una arquitectura distribuida. Las características, las ventajas y los inconvenientes de los sistemas distribuidos vienen descritos en los apartados siguientes.

2.2 ¿Qué es un sistema distribuido?

Un **sistema distribuido** está formado por un conjunto de computadores autónomos unidos por una red de comunicaciones y equipados con software de sistemas distribuidos. El software de sistemas distribuidos permite a los computadores coordinar sus actividades y compartir los recursos del sistema: el hardware, el software y los datos. Los usuarios de un sistema distribuido bien diseñado deberían percibir una única

facilidad de computación, aun cuando dicha facilidad podría estar formada por un conjunto de computadores localizados de manera dispersa.

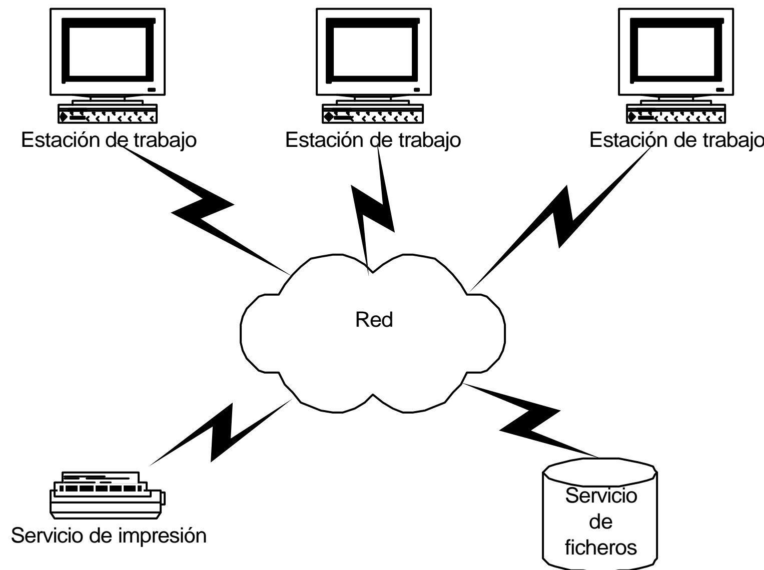


Figura 2.1. Estructura de un sistema distribuido.

El desarrollo de sistemas distribuidos ha seguido la emergencia de las redes de área local de principios de los 70. Más recientemente, la disponibilidad de computadores personales, estaciones de trabajo y servidores de alto rendimiento con un coste relativamente bajo han producido un cambio significativo hacia los sistemas distribuidos, dejando a un lado los grandes sistemas centralizados multiusuario. Esta tendencia se aceleró con el desarrollo de software de sistemas distribuidos, diseñado para permitir el desarrollo de aplicaciones distribuidas.

2.3 Características clave de los sistemas distribuidos

La utilidad de un sistema distribuido viene determinada por la aparición de un **conjunto específico de características** en el mismo: compartición de recursos, apertura (*openness*), flexibilidad, concurrencia, rendimiento, escalabilidad, fiabilidad y tolerancia a fallos, disponibilidad, transparencia, heterogeneidad y seguridad. No se puede asumir que estas características vayan a ser consecuencias propiamente dichas de la distribución, sino todo lo contrario: el software de sistemas y de aplicación habrán de ser diseñados con cuidado con el fin de asegurar que se consiguen todas ellas.

2.3.1 Compartición de recursos

Se utiliza el concepto de **recurso** para referirse a todo aquello que puede ser compartido con éxito en un sistema distribuido. Los beneficios del uso de recursos compartidos ya son bien conocidos de los sistemas operativos centralizados y multiusuario.

- Compartir dispositivos hardware (discos, impresoras y otros periféricos) reduce el coste.
- Compartir datos es esencial en muchas aplicaciones:

- Los equipos de desarrollo de software pueden necesitar acceder al trabajo de los otros y compartir las mismas herramientas de desarrollo con una única copia de compiladores, librerías, editores, depuradores, etc. Cuando se instale una herramienta, todos los usuarios obtienen acceso a ella inmediatamente.
- Muchas aplicaciones comerciales ofrecen a los usuarios la posibilidad de acceder a datos compartidos en una única base de datos activa.
- Un área en rápida expansión para las redes y los sistemas distribuidos es el uso de computadores para soporte de grupos de usuario trabajando de forma cooperativa. Es lo que se conoce como Computer Supported Cooperative Working (CSCW) o groupware y depende en gran medida de los datos que se comparten entre programadores trabajando en distintas estaciones de trabajo.

Los recursos en un sistema distribuido están encapsulados dentro de alguno de los computadores y sólo pueden ser accedidos desde otros computadores utilizando los **mecanismos de comunicación** que se proporcionen para ello. Para que la compartición sea eficiente, cada recurso será gestionado por un programa que ofrecerá una interfaz de comunicación, permitiendo que el recurso sea accedido, manipulado y actualizado de una manera fiable y consistente.

En muchas ocasiones se utiliza el término **gestor de recursos** para nombrar un módulo software que gestiona un conjunto de recursos del mismo tipo. Cada tipo de recurso necesita determinadas políticas y mecanismos de gestión, pero, en general, todos los tipos de recursos tienen algunas necesidades comunes:

- Un **esquema de nombrado** para los recursos que permita acceder a los recursos individuales desde cualquier ubicación.
- La **conversión de nombres** de recursos a direcciones de comunicación.
- La **coordinación de los accesos concurrentes** que cambian el estado de los recursos compartidos, con el fin de asegurar su consistencia.

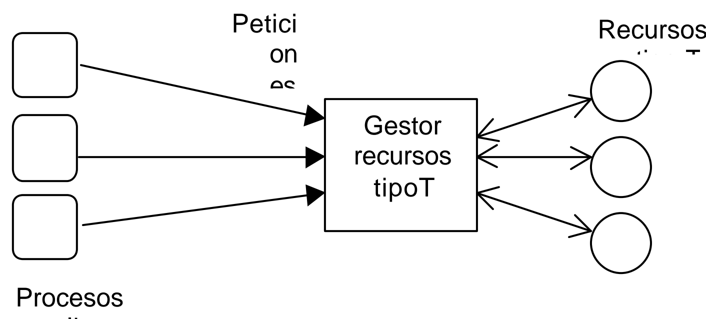


Figura 2.2. Gestor de recursos.

2.3.2 Apertura (del inglés, *openness*)

La apertura de un sistema es una característica que determina si el mismo puede ser extendido de diferentes maneras: hardware y software. Aplicada a un sistema distribuido, esta propiedad viene determinada principalmente por el **grado de permisividad del sistema para añadir servicios** sin duplicar o interrumpir los ya existentes, y se consigue haciendo pública la documentación relativa a las interfaces software consideradas clave en el sistema. Este proceso es semejante a la

estandarización de interfaces, pero no sigue habitualmente los procedimientos oficiales de estandarización, que son lentos y engorrosos.

Históricamente, los sistemas informáticos han sido cerrados. Ejecutaban programas realizados en una serie de lenguajes de programación, pero no permitían a los programadores extender la semántica de los lenguajes para explotar nuevas características del hardware o del sistema operativo.

UNIX [RT74] es el primer ejemplo de un diseño más abierto. Incluye un lenguaje de programación, el C [KR88], que permite a los programadores acceder a todas las facilidades (recursos) gestionadas por el sistema operativo. Dicho acceso se realiza a través de un conjunto de procedimientos, denominados llamadas al sistema, totalmente documentados y disponibles para todos los lenguajes de programación que soporten facilidades de llamada a procedimiento convencionales. Cuando se desea instalar un nuevo periférico en un sistema UNIX, el sistema operativo puede ser extendido para permitir a las aplicaciones acceder a él, añadiendo habitualmente nuevos valores para los parámetros de las llamadas al sistema existentes. Finalmente, al escribirse las aplicaciones en un lenguaje de alto nivel como el C, los programadores pueden realizar programas que se ejecuten sin modificaciones en una amplia variedad de computadores.

Un **sistema distribuido abierto debe ser extensible**, en el sentido de que el conjunto de recursos y facilidades del sistema disponibles para las aplicaciones no esté restringido por las llamadas al sistema del sistema operativo. La **extensibilidad se debe producir a varios niveles**: a nivel hardware, con la adición de computadores a la red, y, a nivel software, con la introducción de nuevos servicios que permitan a los programas de aplicación compartir recursos. Adicionalmente, se cita como beneficio adicional de los sistemas abiertos su **independencia de vendedores individuales**.

2.3.3 Flexibilidad

Otra importante cuestión en el diseño de los sistemas distribuidos es la flexibilidad, que es la característica más importante para el diseño de sistemas abiertos.

La necesidad de flexibilidad viene dada a partir de las siguientes razones:

- **Fácil modificación**. Algunas partes del sistema diseñado van a necesitar ser reemplazadas o modificadas debido a la detección de algún error (*bug*) detectado o porque el diseño ya no es adecuado para el entorno cambiante o los nuevos requerimientos del usuario. Debería ser fácil incorporar cambios en el sistema de forma transparente al usuario.
- **Fácil mejora**. En cualquier sistema, la nueva funcionalidad debe ser añadida según se va solicitando para hacer el sistema más poderoso o fácil de usar. Debería ser fácil añadir nuevos servicios al sistema. Es más, si a un grupo de usuarios no le gusta la forma en que el sistema proporciona un servicio particular, deberían tener la posibilidad de añadir y usar su propio servicio.

El factor de diseño más importante que influye en la flexibilidad de un sistema distribuido es el **modelo usado para diseñar su núcleo** (*kernel*). El núcleo de un sistema opera en un espacio de direcciones inaccesible para los procesos de usuario y es la única parte del sistema que el usuario no puede reemplazar o modificar.

2.3.3.1 Flexibilidad en los sistemas operativos distribuidos

Los dos modelos más comúnmente usados para el diseño del núcleo de los sistemas operativos distribuidos son el núcleo monolítico y el micronúcleo (*microkernel*).

- **Núcleo monolítico**: Mantiene que cada máquina debe ejecutar un núcleo tradicional que proporcione la mayoría de los servicios. Muchos sistemas operativos distribuidos que son extensiones o imitaciones del UNIX usan el modelo monolítico.
- **Micronúcleo**: Sostiene que el núcleo debe proporcionar la menor funcionalidad posible y que el grueso de los servicios debe obtenerse a través de los servidores a nivel usuario.

La segunda corriente es más flexible ya que el sistema operativo sólo proporciona cuatro servicios mínimos: mecanismos de comunicación entre procesos, administración de memoria a bajo nivel, administración de procesos a bajo nivel y gestión de la entrada/salida.

Comparando ambos modelos, el modelo **micronúcleo tiene varias ventajas**. En primer lugar, el tamaño de los núcleos monolíticos proporciona flexibilidad y configurabilidad menores.

El modelo micronúcleo, modular por naturaleza, es fácil de diseñar, implementar e instalar y, dado que muchos servicios se implementan a nivel usuario, es fácil de modificar el diseño de un servicio o añadir un servicio nuevo para lo que no es necesario parar el sistema y arrancar un nuevo núcleo.

La única ventaja potencial de la primera opción es el rendimiento, ya que las llamadas al núcleo y la realización por parte de éste de todo el trabajo puede ser más rápido que el envío de mensajes a los servidores remotos.

A pesar de la potencial pérdida de rendimiento, el modelo **micronúcleo es el preferido para el diseño de sistemas operativos modernos** por las siguientes razones:

- Las ventajas de flexibilidad del modelo micronúcleo compensan la penalización del rendimiento
- Algunos resultados experimentales han mostrado que, aunque en la teoría el modelo micronúcleo parece tener un rendimiento más pobre que el modelo monolítico, en la práctica otros factores tienden a dominar.

2.3.4 Concurrencia

Los sistemas distribuidos **amplían la posibilidad de ejecución concurrente** de los sistemas centralizados (ejecución paralela, en el caso de que dispongan de varios procesadores) por el hecho de existir varios computadores, cada uno de los cuales con uno o varios procesadores.

Si existe un único procesador, la concurrencia entre varios procesos de una máquina se consigue intercalando la ejecución de trozos de los procesos. Si hay N procesadores, se pueden ejecutar en paralelo hasta N procesos.

En un sistema distribuido existen muchos computadores, cada una con uno o más procesadores. Si hay M computadores con un procesador cada uno, entonces pueden ejecutarse en paralelo hasta M procesos, suponiendo que los procesos estén situados en máquinas distintas.

En un sistema distribuido basado en el modelo de compartición de recursos descrito, las oportunidades de ejecución paralela se basan en dos razones:

- Muchos usuarios invocan simultáneamente comandos o interaccionan con programas de aplicación. De esta forma, uno o más procesos de aplicación se ejecutan en nombre de cada usuario activo. En la mayoría de los sistemas distribuidos, los procesos de aplicación se ejecutan en la estación de trabajo del usuario y no entran en conflicto por los recursos de procesamiento con los procesos de aplicación de otros usuarios. Si la estación de trabajo tiene un único procesador y hay más de un proceso de aplicación ejecutándose en ella, se ejecutan de forma intercalada. Las estaciones de trabajo con múltiples procesadores hacen posible ejecutar varias aplicaciones en paralelo o ejecutar aplicaciones que explotan más de un procesador.
- Muchos procesos servidores se ejecutan concurrentemente y cada uno responde a diferentes peticiones de distintos clientes. Puede existir más de un proceso servidor para cada tipo de recurso. Normalmente se ejecutan en computadores distintos, haciendo posible que cada servidor se ejecute en paralelo con los otros y con los procesos que se ejecutan en estaciones de trabajo. Las peticiones para acceder a los recursos en un servidor determinado se encolan al servidor y se procesan en secuencia. Incluso varias peticiones pueden ser procesadas concurrentemente por alguna de las múltiples instancias de los procesos gestores de recursos. Cuando varios procesos acceden al mismo recurso concurrentemente, el servidor debe sincronizar las acciones para asegurarse de que no hay conflicto. La sincronización debe ser planeada cuidadosamente para asegurar que los beneficios de la concurrencia no se pierden.

Las peticiones que llegan a un gestor de recursos deben ser procesadas adecuadamente para garantizar un acceso consistente a los mismos. Una posibilidad es ir encolando las peticiones y tratándolas de manera secuencial. Otra posibilidad es crear una instancia del gestor de recursos para cada petición, instancias que tendrán que sincronizar sus acciones para no entrar en conflicto.

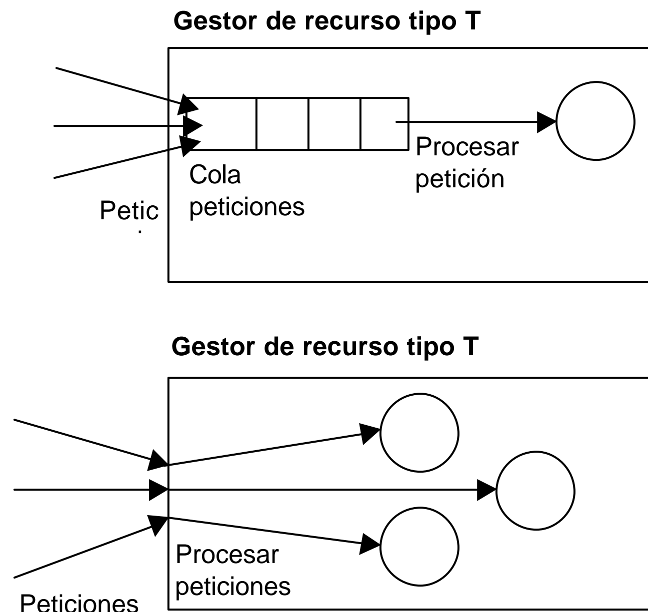


Figura 2.3. Distintas formas de gestión de peticiones sobre recursos.

2.3.5 Rendimiento

La ejecución de una aplicación en un sistema distribuido debe ser mejor (o al menos, no debe ser peor) que su ejecución en un único procesador.

El problema del rendimiento se complica por el hecho de que la comunicación, factor esencial en un sistema distribuido (y ausente en un sistema centralizado) es lenta, en general. El envío de un mensaje y la obtención de una respuesta puede tardar un milisegundo en una red de área local (LAN, *Local Area Network*). Para mejorar el rendimiento, parece que hay que minimizar el número de mensajes. El problema de esta estrategia es que la mejor forma de mejorar el rendimiento es tener muchas actividades ejecutándose de manera paralela en distintos procesadores, lo que implica el envío de muchos mensajes.

Una posible solución es dividir los trabajos utilizando **paralelismo de grano fino** (muchas tareas con poco trabajo) o **paralelismo de grano grueso** (pocas tareas con mucho trabajo). Dado que las tareas del primer modelo se comunicarán con más frecuencia, para que el paralelismo de grano grueso se ajusta mejor al modelo comentado.

Otra fuente de problemas es la tolerancia a fallos (que se describe más adelante en este capítulo), ya que la redundancia de servidores implica también una redundancia de mensajes.

2.3.6 Escalabilidad

Los sistemas distribuidos deben operar de una manera efectiva a **diferentes escalas**: desde el sistema distribuido mínimo, consistente en dos computadores y un servidor de ficheros, pasando por redes de área local que contienen un conjunto apreciable de computadores y servidores especializados, hasta conjuntos de redes locales interconectadas que contienen cientos de computadores. En todos estos casos, el conjunto de computadores forma un único sistema distribuido que permite la compartición de los recursos entre todos ellos.

El **software de sistemas y de aplicación no deben cambiar** cuando la escala del sistema cambia.

La necesidad de escalabilidad no es sólo un problema hardware o de rendimiento de red. Afecta al diseño de casi cualquier aspecto de un sistema distribuido. En un sistema centralizado ciertos recursos compartidos (memoria, procesadores, canales de entrada/salida) se suministran de forma limitada y no pueden ser replicados infinitamente. En los sistemas distribuidos, la limitación en el suministro de algunos de esos recursos se elimina, ya que es posible añadir, por ejemplo, más computadores o más memoria. Pero permanecen otras limitaciones si el diseño del sistema no reconoce la necesidad de escalabilidad. Dichas limitaciones se producen más bien en el ámbito del software de sistemas distribuidos.

La demanda de escalabilidad en los sistemas distribuidos ha llevado a una política de diseño en la que se asume que **ningún recurso hardware o software puede ser suministrado de forma restrictiva**. En lugar de eso, como la demanda de recursos crece, debería ser posible extender el sistema para cubrirla.

A medida que los sistemas distribuidos se hacen más grandes, se perfilan los siguientes principios de diseño: evitar componentes y algoritmos centralizados y ejecutar el máximo de operaciones en las estaciones de trabajo de los usuarios.

Evitar componente centralizados

En el diseño de un sistema operativo distribuido, el uso de componentes centralizados como un servidor de ficheros o una base de datos para el sistema completo hace que el sistema no sea escalable debido a las siguientes razones:

- Normalmente, el fallo de una entidad centralizada hace que todo el sistema caiga.
- El rendimiento de la entidad centralizada se convierte normalmente en un cuello de botella.
- Incluso en el caso de que una entidad centralizada tenga suficiente capacidad de procesamiento y almacenamiento, la capacidad de la red que conecta la entidad centralizada con otros nodos del sistema se satura frecuentemente cuando la contención por la entidad se incrementa más allá de un nivel determinado.
- En una gran red formada por varias redes de área local interconectadas, es obviamente ineficiente que un servicio particular esté siempre servido por un servidor que está a varias redes de distancia.

La replicación de recursos y los algoritmos de control distribuido son técnicas de uso frecuente para conseguir este objetivo. De hecho, para mejorar la escalabilidad debería utilizarse una configuración simétrica en la que todos los nodos del sistema sean aproximadamente iguales.

Evitar algoritmos centralizados

Un algoritmo centralizado es aquel que opera recopilando información de todos los nodos, procesándola en un único nodo y distribuyendo los resultados a los otros. El uso de tales algoritmos en el diseño de un sistema operativo distribuido no es aceptable desde el punto de vista de la escalabilidad. Las razones para ello, son muy similares a las vistas para los componentes.

Ejecutar el máximo de operaciones en las estaciones de trabajo de los clientes

Cuando sea posible, una operación debería ejecutarse en la propia estación de trabajo del cliente en lugar de una máquina servidora. Esto es debido a que el servidor es un recurso para varios clientes. Este principio mejora la escalabilidad y permite minimizar la degradación del rendimiento del sistema a medida que este crece reduciendo la contención por los recursos compartidos.

2.3.7 Fiabilidad y tolerancia a fallos

Este es uno de los objetivos originales de la construcción de sistemas distribuidos, el hacerlos más fiables que los sistemas con un único procesador, debido precisamente a la existencia de múltiples instancias de los recursos. Sin embargo, el sistema operativo debe ser diseñado de forma apropiada para sacar provecho de esta característica de un sistema distribuido.

Un **fallo** es un defecto algorítmico o mecánico que puede generar un error y que causa un fallo del sistema. Dependiendo del modo en que se comporta el sistema en caso de fallo, los fallos del sistema son de dos tipos:

- ***Fail stop***: El sistema detiene su funcionamiento después de cambiar a un estado en el que el fallo puede ser detectado.
- **Bizantino**: El sistema sigue funcionando pero producirá resultados erróneos.

Para que un sistema sea fiable, los mecanismos de gestión de fallos del sistema operativo deben diseñarse de tal forma que evite, tolere y detecte y recupere los fallos.

2.3.7.1 Evitar fallos

Implica el diseño del sistema de tal forma que la ocurrencia de fallos se minimice.

2.3.7.2 Tolerar fallos

La tolerancia a fallos es la habilidad de un sistema de continuar funcionando en caso de fallos parciales del sistema. El rendimiento del sistema podría degradarse pero el sistema seguiría funcionando de forma adecuada. Algunos conceptos importantes que pueden usarse para mejorar la tolerancia a fallos de un sistema operativo distribuido son:

Técnicas de redundancia

Se trata de evitar puntos de fallo en el sistema replicando el hardware y el software críticos de tal forma que, si una instancia falla, las otras pueden seguir siendo utilizadas. Con estas técnicas, se introduce cierta sobrecarga al sistema para mantener consistentes dos o más copias del recurso. Esto plantea una cuestión importante acerca de qué cantidad de réplicas es suficiente y no demasiada.

Control distribuido

Para una mejor fiabilidad, muchos de los algoritmos y protocolos utilizados en un sistema operativo distribuido deben emplear un mecanismo de control distribuido para evitar puntos únicos de fallo. Es importante recalcar que, cuando se utilizan múltiples servidores distribuidos en un sistema para proporcionar un tipo de servicio, los servidores deben ser independientes. Es decir, el diseño no debe requerir el funcionamiento simultáneo de los servidores.

2.3.7.3 Detección y recuperación de fallos

La detección y recuperación de fallos para mejorar la fiabilidad implican el uso de mecanismos hardware y software para determinar la existencia de un fallo y corregirlo, llevando al sistema a un estado aceptable para continuar su ejecución. Algunas de las técnicas más usadas son:

Transacciones atómicas

Una transacción atómica es una computación que consiste en una serie de operaciones que tienen lugar de forma indivisible en presencia de fallos y de otras computaciones concurrentes. Es decir, o todas las operaciones se ejecutan con éxito o ninguna tiene efecto y ningún otro proceso en ejecución concurrente puede modificar o siquiera observar los estados intermedios de las operaciones. Las transacciones ayudan a preservar la consistencia de un conjunto de objetos compartidos in presencia de fallos y de accesos concurrentes. Hacen la recuperación mucho más facil debido a que una transacción finaliza sólo en uno de dos estados: o todo ha ido bien o ninguna de las operaciones se ha ejecutado.

En un sistema con transacciones, si un proceso se para de forma inesperada debido a un fallo hardware o a un error software antes de que la transacción termine, el sistema restaura los objetos que se estuviesen ejecutando a sus estados originales. Si el sistema no soporta transacciones, fallos inesperados de un proceso durante el procesamiento de una operación pueden dejar los objetos bajo modificación en un estado inconsistente. Es

más, sin transacciones puede ser difícil o imposible recuperar los objetos a sus estados originales.

Servidores sin estado

En el modelo cliente-servidor, utilizado frecuentemente en sistemas distribuidos para servir las solicitudes de usuario, un servidor puede implementarse usando uno de los siguientes paradigmas de servicio: con o sin estado. Estos paradigmas se distinguen por un aspecto de la relación cliente-servidor: si la historia de la solicitud entre cliente y servidor afecta o no a la ejecución del siguiente servicio solicitado.

La aproximación con estado depende de la historia de las peticiones servidas, mientras la aproximación sin estado no. Los servidores sin estado tienen una ventaja sobre los servidores con estado en caso de fallo. El paradigma de servicio sin estado hace que la recuperación ante fallos sea mucho más sencilla, dado que no se mantiene ninguna información en el servidor acerca del estado del cliente. Por otro lado, el paradigma del servidor con estado requiere un procedimiento completo de recuperación ante fallos. Tanto el cliente como el servidor necesitan detectar fallos. El servidor necesita detectar las caídas del cliente para poder descartar cualquier estado que se esté manteniendo sobre el cliente y el cliente debe detectar las caídas del servidor para poder realizar las funciones de manejo de errores necesarias.

Retransmisión de mensajes basada en temporizadores y reconocimiento

En un sistema distribuido, eventos como el fallo de un nodo o la caída de una línea de comunicación pueden interrumpir una comunicación en marcha entre dos procesos provocando pérdidas de mensajes. Un mecanismo de comunicación fiable debe tener mecanismos de detección de pérdida de mensajes para poder retransmitirlos. La gestión de los mensajes perdidos implica, habitualmente, el retorno de mensajes de reconocimiento (*acknowledgement*) y la retransmisión en caso de vencimiento de algún temporizador. Es decir, el receptor debe retornar un mensaje de reconocimiento en un intervalo de tiempo y, en caso de que en ese tiempo el remitente no reciba el reconocimiento, asume que el mensaje se ha perdido y lo retransmite. Uno de los mayores problemas asociados con esta aproximación es el de la duplicación de mensajes que puede producirse al retransmitir un mensaje en caso de vencimiento del temporizador. Un mecanismo de comunicación fiable debería ser capaz de detectar y gestionar mensajes duplicados. La gestión de mensajes duplicados implica normalmente un mecanismo para generar y asignar automáticamente números a los mensajes.

Cualquier mecanismo utilizado para incrementar la fiabilidad del sistema distribuido provoca una potencial pérdida de eficiencia debido a la sobrecarga que introducen estas técnicas. Uno de los mayores objetivos en el diseño de sistemas operativos distribuidos consiste en integrar estos mecanismos de forma que el coste que introduzcan sea aceptable.

2.3.8 Disponibilidad

Los sistemas distribuidos también proporcionan un alto grado de disponibilidad en el caso de fallos hardware. La disponibilidad de un sistema es una medida de la **proporción de tiempo que está disponible** para su uso. Un fallo en un computador multi-usuario casi siempre provoca que deje de estar disponible para todos sus usuarios. Cuando uno de los componentes en un sistema distribuido falla, sólo el trabajo que está usando el componente que falla se ve afectado. Por ejemplo, un usuario puede moverse

a otra estación de trabajo si la que está usando falla. De una forma similar, un proceso servidor puede ser rearrancado en otra máquina.

2.3.9 Transparencia

La transparencia se define como el **ocultamiento** al usuario del hecho de que los componentes de un sistema distribuido están dispersos y separados, de tal forma que el sistema se perciba como un todo único en lugar de una colección independiente de componentes.

La separación de componentes es inherente a los sistemas distribuidos. Esta separación posibilita la ejecución paralela de programas, la contención y recuperación de fallos en los componentes sin que afecte al resto del sistema, el uso de canales de comunicación controlados y aislados como un método para reforzar la seguridad y la protección y el crecimiento/disminución incremental del sistema a través de la adición/eliminación de componentes.

Las consecuencias de la **transparencia tienen una gran influencia en el diseño** del sistema.

La **transparencia es la clave y el objetivo de los sistemas operativos distribuidos**. Un sistema operativo distribuido debe estar diseñado de tal forma que una colección de máquinas distintas conectadas por un sistema de comunicación, se presente ante los usuarios como un uniprocesador. El usuario no debe tener conocimiento de dónde están localizados sus ficheros o dónde se están ejecutando sus trabajos, es decir, se trata de engañar al usuario de forma que para éste todo el sistema parezca una única máquina con un sólo procesador en tiempo compartido.

El *ANSA Reference Manual* [ANS89] y el *International Standars Organization´s Reference Model for Open Distributed Processing (RM-ODP)* [ISO92] identifican ocho formas de transparencia, que proporcionan un resumen de las motivaciones y objetivos de los sistemas distribuidos. En estas definiciones *ANSA* utiliza el término **objeto de información** para denotar las entidades a las que se aplica la transparencia de distribución.

Transparencia de acceso

Permite el acceso a los objetos de información usando un conjunto de operaciones común, independientemente de que éstos sean locales o remotos.

Los usuarios no necesitan saber si un recurso es local o remoto. Esto significa que el sistema operativo distribuido debería permitir a los usuarios el acceso a recursos remotos del mismo modo que a recursos locales. La interfaz de usuario o llamadas al sistema no debería distinguir entre recursos locales o remotos y debe ser responsabilidad del sistema operativo localizar los recursos y arreglarlo todo para dar servicio a las peticiones de usuario. Un claro ejemplo de falta de transparencia de acceso es el presentado por un conjunto de máquinas UNIX unidas mediante una red en las que el acceso a ficheros locales se realiza mediante una llamada al sistema operativo local (*open, read*, …) mientras que los accesos a ficheros remotos necesitan una serie de operaciones especiales (*ftp*, …) para traer una copia al disco duro local.

Transparencia de localización

Permite acceder a objetos de información sin conocimiento de su localización. Los usuarios no deben ni pueden indicar en sus operaciones la localización de los recursos

del sistema, tanto software como hardware (CPU, ficheros, etc.). Por tanto, la dirección del recurso no está codificada en el nombre del recurso.

Los dos aspectos principales en la transparencia de localización son:

- **Transparencia de nombres**: Se refiere al hecho de que un recurso no revela su localización física, es decir, el nombre de un recurso es independiente de la topología del sistema, la conexión o la localización actual del recurso. Es más, para todos aquellos recursos que puedan moverse por el sistema (como ficheros) debe estar permitido moverlos sin que sean necesarios cambios en los nombres. Los nombres deben ser únicos en el sistema.
- **Movilidad de usuario**: Se refiere al hecho de que independientemente de la máquina en que se conecta el usuario, debe tener acceso a cualquier recurso con el mismo nombre. Es decir, el usuario no debería tener que usar distintos nombres para acceder a un mismo recurso desde dos nodos distintos.

Los dos tipos de transparencia más importantes son las de acceso y localización. Su presencia o ausencia afecta de una manera muy importante al uso de los recursos distribuidos. Se utiliza habitualmente el término **transparencia de red** para referirse a ambas simultáneamente. La transparencia de red proporciona un grado de anonimato para los recursos similar al encontrado en los sistemas operativos centralizados.

Transparencia de concurrencia

Permite que varios procesos operen concurrentemente utilizando objetos de información compartidos, sin que por ello haya interferencias entre ellos.

En un sistema distribuido, existen múltiples usuarios separados en el espacio que usan el sistema concurrentemente. En tal situación, es necesario permitir que se compartan recursos entre los procesos en ejecución. Sin embargo, y debido a que el número de recursos en el sistema está restringido, un proceso influirá en la ejecución de otro que se ejecuta concurrentemente al necesitar competir por los recursos. La transparencia de concurrencia significa que cada usuario tiene la impresión de que es el único usuario del sistema. Para proporcionar transparencia de concurrencia, los mecanismos de compartición de recursos deben tener las siguientes propiedades:

- **Propiedad de ordenación de eventos**: asegura que todas las peticiones de acceso a los recursos del sistema se ordenan de forma apropiada para proporcionar una visión consistente a los usuarios.
- **Propiedad de exclusión mutua**: asegura que, en cualquier momento, como mucho un proceso accede a un recurso compartido que no debe ser usado simultáneamente por varios procesos si la operación del programa debe ser correcta.
- **Propiedad de no-inanición**: asegura que en caso de que un recurso no pueda ser utilizado simultáneamente por varios procesos, cualquier petición lo liberará en algún momento y se le concederá a otro.
- **Propiedad de no interbloqueo**.

Transparencia de replicación

Permite el uso de múltiples instancias de objetos de información para aumentar la fiabilidad y el rendimiento, sin que los usuarios o los programas de aplicación tengan conocimiento de la existencia de las réplicas.

Para un mejor rendimiento y fiabilidad, casi todos los sistemas operativos distribuidos permiten la creación de réplicas de recursos en distintos nodos del sistema. En estos sistemas, la existencia de múltiples copias de un recurso y la actividad de replicación deberían ser transparentes al usuario. Dos cuestiones importantes relacionadas con la transparencia de replicación son el nombrado de las réplicas y el control de la replicación. Es responsabilidad del sistema operativo dar nombre a las copias del recurso y establecer la correspondencia entre el nombre proporcionado por el usuario y una réplica apropiada del recurso. Es más, las decisiones acerca del control de la replicación tales como cuántas copias de cada recurso deberían crearse, dónde debería estar colocada cada réplica y cuándo debería crearse o borrarse cada una deben estar automatizadas en el sistema.

Transparencia de fallo

Oculta los fallos, permitiendo a los usuarios y programas de aplicación completar las tareas a pesar de fallos en componentes hardware o software.

El sistema operativo debe enmascarar al usuario los fallos parciales del sistema como un fallo en la comunicación, el fallo de una máquina o de un dispositivo de almacenamiento. Un sistema operativo distribuido que soporte esta propiedad continuará funcionando en presencia de fallos. La transparencia completa frente a fallos no es alcanzable hoy día con el estado del arte actual en los sistemas operativos distribuidos dado que no todos los tipos de fallos pueden ser gestionados de forma transparente al usuario. Es más, cualquier intento de construir un sistema operativo distribuido completamente transparente a fallos resultaría muy lento y costoso debido a la gran cantidad de información redundante requerida para gestionar cualquier tipo de fallos.

Transparencia de migración

Permite el movimiento de objetos de información dentro del sistema sin que esto afecte a las operaciones de los usuarios o programas de aplicación.

Por razones de rendimiento, fiabilidad y seguridad, aquellos objetos que tienen capacidad de moverse, son migrados frecuentemente de un nodo a otro del sistema distribuido. El objetivo de la transparencia de migración es asegurar que el movimiento del objeto es gestionado automáticamente por el sistema de forma transparente al usuario. Hay tres cuestiones importantes para conseguir este objetivo:

- Las decisiones de migración tales como qué objeto mover desde dónde hasta dónde, debe llevarlas a cabo el sistema automáticamente.
- La migración de un objeto de un nodo a otro no debería requerir ningún cambio en su nombre.
- Cuando el objeto que migra es un proceso, el mecanismo de comunicación entre procesos debería asegurar que cualquier mensaje enviado a ese proceso llegará a su destino sin necesidad de que el proceso remitente lo reenvíe si el receptor se mueve mientras el mensaje está siendo enviado.

Transparencia de rendimiento

Permite que el sistema sea reconfigurado para mejorar el rendimiento cuando la carga varía.

Aquella situación en la que un procesador del sistema esté sobrecargado mientras otro está ocioso no debe ocurrir, es decir, la capacidad de procesamiento del sistema debería distribuirse uniformemente entre todos los trabajos disponibles en el sistema.

Transparencia de escalabilidad

Permite que el sistema y las aplicaciones se expandan sin tener que cambiar la estructura del sistema o los algoritmos de la aplicación. Esto implica que la arquitectura debe ser diseñada siguiendo criterios de sistemas abiertos y los algoritmos deben ser escalables.

2.3.10 Heterogeneidad

Un sistema distribuido heterogéneo consiste en conjuntos interconectados de hardware o software distintos. Dada esta diversidad, el diseño de sistemas distribuidos heterogéneos es más difícil que en el caso homogéneo donde cada sistema está basado en el mismo o similar hardware y software.

En un sistema heterogéneo es necesario algún tipo de transformación de datos para interaccionar entre dos nodos incompatibles. Algunos de los primeros sistemas dejaban esta transformación a los usuarios, hecho inaceptable. Esta transformación puede llevarse a cabo tanto en el lado del cliente como del servidor. Tanto uno como otro tienen serios inconvenientes.

La complejidad de este proceso de transformación puede reducirse utilizando un **formato intermedio estándar** para los datos . Con esta solución se declara un formato intermedio y cada nodo sólo requiere software que le permita transformar su propio formato al estándar y viceversa.

2.3.11 Seguridad

Con el fin de que los usuarios puedan confiar en el sistema, es necesario proteger los distintos recursos del mismo frente a accesos no autorizados. Asegurar la seguridad en un sistema distribuido es más difícil que en uno centralizado dada la inexistencia de un punto de control único y el uso de redes inseguras para el tráfico de datos.

En un sistema centralizado todos los usuarios son autenticados por el sistema en el proceso de conexión (*login*) y es sencillo comprobar si el usuario está autorizado o no.

En un sistema distribuido, sin embargo, dado que el modelo cliente/servidor se usa frecuentemente para solicitar y servir peticiones, cuando un cliente envía un mensaje al servidor, el servidor debe saber quién es el cliente. Esto no es tan simple como enviar con el mensaje la identificación del remitente, porque algún intruso podría pretender pasar por un cliente autorizado o podría cambiar el mensaje durante la transmisión. La seguridad en un sistema distribuido tiene los siguientes requerimientos adicionales:

- Debe ser posible que el remitente de un mensaje sepa si el mensaje fue recibido por el receptor deseado.
- Debe ser posible que el receptor de un mensaje sepa si el mensaje fue enviado por el remitente indicado.
- Debe ser posible que el remitente y el receptor estén seguros de que el contenido de un mensaje no fue cambiado durante la transferencia de datos.

2.4 Ventajas e inconvenientes de los sistemas distribuidos

Los sistemas operativos distribuidos son mucho más complejos de construir que los centralizados debido al hecho de que el sistema, además de tener que usar y gestionar los recursos distribuidos, también debe ser capaz de gestionar la comunicación y la seguridad. A pesar de esto, el uso de los sistemas distribuidos tiene un ritmo de crecimiento rápido.

2.4.1 Ventajas de los sistemas distribuidos frente a los centralizados

2.4.1.1 Distribución inherente de las aplicaciones

Un elemento a favor de los sistemas distribuidos es la naturaleza inherentemente distribuida de algunas aplicaciones, como las de los bancos en las que se trabaja sobre bases de datos comunes en diferentes sucursales, y en general aplicaciones que utilizan máquinas que están separadas una cierta distancia, reservas de líneas aéreas, etc.

2.4.1.2 Compartición de datos y dispositivos

La necesidad de compartir datos entre máquinas y usuarios diferentes, pudiendo utilizar ficheros compartidos es también uno de los principales motivos de la aparición de los sistemas distribuidos. Por otro lado la compartición de dispositivos físicos, como las impresoras, aumenta considerablemente las prestaciones.

2.4.1.3 Economía

En primer lugar los sistemas distribuidos tienen una razón precio/rendimiento mucho mejor que la de un único sistema centralizado. Este hecho se empezó a notar con la aparición de la tecnología de los microprocesadores, que dejan a un lado los grandes computadores de los años ochenta. Así, si se desea una gran capacidad de cómputo resulta mucho más económico comprar un conjunto de procesadores reunidos en un mismo sistema que un único procesador muy potente. Esta es la principal razón de la tendencia hacia los sistemas distribuidos.

2.4.1.4 Velocidad

La velocidad de una máquina *mainframe* está restringida a un cierto límite. Sin embargo, la utilización en paralelo de múltiples procesadores hacen que la velocidad que se pueda alcanzar sea mucho mayor. Es más, si es posible partir una computación en subcomputaciones que se ejecuten concurrentemente en los distintos procesadores, la mejora sería considerable.

2.4.1.5 Fiabilidad

Los sistemas distribuidos aportan una mayor fiabilidad que los centralizados al distribuir la carga entre muchos procesadores; el fallo de uno de ellos no tiene por qué, provocar fallos en el resto. Con un software adecuado el resto de las máquinas podrían encargarse de su trabajo.

Un aspecto muy importante de la fiabilidad es la disponibilidad que se refiere a la fracción de tiempo durante la cual el sistema está disponible para su uso. En comparación con un sistema centralizado, un sistema distribuido tiene un grado de disponibilidad mayor, lo que es importante en la computación de aquellas aplicaciones críticas cuyo fallo puede ser desastroso.

Sin embargo, la fiabilidad suele implicar una penalización en el rendimiento. Es necesario mantener una solución de compromiso entre ambos.

2.4.1.6 Extensibilidad y crecimiento por incrementos

Los sistemas distribuidos son más receptivos al crecimiento ya que es fácil añadir nuevos procesadores con poco coste si el sistema actual se queda pequeño.

2.4.1.7 Flexibilidad con respeto a la utilización de una máquina aislada

El usuario puede tener a su disposición un conjunto de ordenadores que podrán repartir la carga de trabajo de manera adecuada para conseguir una computación más eficaz. Además, al tener disponible un conjunto de máquinas distintas entre las que se puede seleccionar la más adecuada para procesar el trabajo del usuario dependiendo de la naturaleza del trabajo.

2.4.2 Desventajas de los sistemas distribuidos

2.4.2.1 Software

El problema fundamental de los sistemas distribuidos es el software. Aún no existe mucha experiencia en el diseño, implantación y uso de software distribuido. Precisamente, este es un campo de investigación actual.

2.4.2.2 Redes

Las redes son indispensables para la comunicación entre máquinas, sin embargo, pueden plantear problemas de saturación o pérdidas de mensajes.

2.4.2.3 Seguridad

El posible acceso a todo el sistema por parte de los usuarios plantea el inconveniente de la necesidad de un sistema de seguridad adecuado.

A pesar de todos estos problemas, la tendencia hacia la distribución es indiscutible.

2.5 Definición y funciones de un sistema operativo distribuido

En los apartados anteriores se describieron con detalle las características, ventajas e inconvenientes de la utilización de sistemas distribuidos. Llega el momento de definir lo que se entiende por **sistema operativo distribuido**.

En primer lugar, un sistema operativo distribuido es un tipo de sistema operativo. En este sentido, un sistema operativo distribuido proporciona un entorno para la ejecución de los programas de usuario, controla su ejecución para prevenir errores y gestiona los recursos disponibles.

En segundo lugar, un sistema operativo distribuido es un tipo de sistema distribuido. Por tanto, todas las características anteriormente descritas le son aplicables. Evidentemente, no se exige a todo sistema operativo distribuido todas las características anteriores con un grado de exigencia máximo, aunque sí hay una característica en la que la mayor parte de los autores suelen hacer hincapié: la transparencia.

Teniendo en cuenta todo lo dicho anteriormente, es posible dar una definición informal de un sistema operativo distribuido [Hut87]:

> *"Un sistema operativo distribuido es aquel que proporciona a sus usuarios la misma visión que un sistema operativo centralizado ordinario, pero que se ejecuta en unidades de procesamiento múltiples e independientes. El concepto clave es la transparencia. En otras palabras, el uso de múltiples procesadores debería ser invisible (transparente) al usuario"*

En la literatura de sistemas operativos se tiende a definir el concepto de sistema operativo ofreciendo una descripción de su funcionalidad. Siguiendo dicha tendencia, y con el fin de completar la definición de sistema operativo distribuido, se puede decir que un **sistema operativo distribuido** debe:

- Controlar la asignación de recursos para posibilitar su uso de la manera más eficiente. Debido al carácter distribuido de los recursos, el sistema operativo podrá optar por la utilización de técnicas de replicación, que permitan acercar al máximo los recursos a las aplicaciones que los utilizan.
- Proporcionar al usuario un computador virtual que le sirva como entorno de programación de alto nivel. No es conveniente que el usuario, al menos el poco experimentado, tenga que lidiar con la complejidad del sistema distribuido. Al igual que ocurrió en su momento con la introducción de los sistemas de tiempo compartido, se proporcionará al usuario un computador virtual donde escribir y ejecutar sus programas.
- Ocultar la distribución de los recursos. El computador virtual que se proporciona a los usuarios estará formado por un conjunto de recursos virtuales, que el usuario podrá manejar como si fueran locales. El sistema operativo se encargará de la asociación de los recursos virtuales con recursos reales, con la dificultad añadida de que pueden encontrarse en cualquier lugar del sistema distribuido.
- Proporcionar mecanismos para proteger los recursos del sistema contra accesos de usuarios no autorizados. Los recursos del sistema operativo, tomen la forma que tomen, deberán ser protegidos de la misma forma que ocurría en los sistemas centralizados. El sistema operativo tendrá en cuenta que las peticiones de utilización de dichos recursos pueden llegar de cualquier parte del sistema distribuido.
- Proporcionar una comunicación segura. En un sistema distribuido las comunicaciones son una parte imprescindible. A través de las líneas de comunicación que unen los diferentes nodos del sistema distribuido circula información crítica para el buen funcionamiento del sistema. Es imprescindible que dicha información está a salvo de lecturas o modificaciones por parte de usuarios no autorizados que podrían traer consigo un incorrecto funcionamiento del sistema o la obtención por parte los usuarios de privilegios para su utilización.

El diseño de un sistema operativo distribuido es más complejo que el diseño de un sistema operativo centralizado por varias razones. En el diseño de un sistema operativo centralizado se asume que el sistema operativo tiene completo acceso a información acerca del sistema y su funcionamiento. Sin embargo, un sistema operativo distribuido, debe diseñarse con la suposición de que tal información completa y centralizada nunca estará disponible. En un sistema distribuido los recursos están físicamente distribuidos, no existe un reloj centralizado en el sistema, el envío de mensajes supone un retraso en el sistema e incluso, pueden perderse. Debido a todas estas razones, un sistema operativo distribuido no puede tener una información global, actualizada y consistente acerca del estado de los distintos componentes del sistema.

2.6 Resumen

Los sistemas distribuidos se están convirtiendo en la norma habitual para la organización de sistemas informáticos, en detrimento de los sistemas centralizados convencionales. Se están utilizando cada vez más en la construcción de todo tipo de aplicaciones, donde las comunicaciones se presentan ya como un requerimiento básico.

En los diferentes ámbitos en los que se pueden aplicar, los sistemas distribuidos proporcionan beneficios sustanciales a sus usuarios. Ahora bien, para proporcionar dichos beneficios, es necesario que un sistema distribuido esté caracterizado por el siguiente conjunto de propiedades: compartición de recursos, apertura (openness), flexibilidad, concurrencia, rendimiento, escalabilidad, fiabilidad y tolerancia a fallos, disponibilidad, transparencia, heterogeneidad y seguridad. Dichas propiedades no son consecuencias directas de la distribución, sino que el sistema distribuido habrá de ser diseñado y construido con cuidado para conseguirlas. De entre todas ellas, destacan fundamentalmente la compartición de recursos, la transparencia y la flexibilidad.

Los sistemas operativos distribuidos son un caso particular de sistemas distribuidos. Tratan de proporcionar la funcionalidad de los sistemas operativos tradicionales ejecutándose en un entorno distribuido. En el diseño de un sistema operativo distribuido cobra más importancia, si cabe, la transparencia. Es imprescindible proporcionar al usuario la ilusión de una única facilidad de computación, en la que el conjunto distribuido de recursos se muestra como un computador virtual en el que el usuario ejecutará sus aplicaciones.

CAPÍTULO 3 REQUISITOS DE UN SISTEMA DE DISTRIBUCIÓN DE OBJETOS

3.1 Introducción

Una vez que se han definido las características de los sistemas distribuidos es necesario plantearse la forma en que se va a afrontar su diseño y construcción. Habitualmente, los sistemas distribuidos se han venido construyendo utilizando conceptos utilizados en los sistemas operativos tradicionales. El concepto de proceso y los mecanismos de comunicación entre procesos son los más destacables.

En los últimos tiempos, el auge de las tecnologías orientadas a objetos ha llevado a los diseñadores a plantearse la forma en que pueden aplicarlas para la construcción de sistemas distribuidos. La aplicación de dichas tecnologías no se ha detenido en la construcción en sí del sistema distribuido, sino en la forma que éste proporciona sus servicios y en la manera en que se construyen las aplicaciones de usuario.

3.2 Arquitectura de sistemas distribuidos

En el capítulo anterior se describieron las características esenciales de todo sistema distribuido. El problema que surge inmediatamente es cómo construir el sistema distribuido para que cubra dichas expectativas. La construcción se deberá realizar basándose en uno de los dos modelos siguientes: el modelo de procesos (o cliente/servidor) o el modelo de objetos.

3.2.1 Modelo de procesos o cliente/servidor

Es el más ampliamente conocido y aceptado para los sistemas distribuidos tradicionales. Existe un conjunto de **procesos servidores**, cada uno de los cuales actúa como **gestor de recursos** para una colección de recursos de un tipo dado y una colección de **procesos clientes**, cada uno de los cuales ejecuta una tarea que requiere acceder a algunos de los recursos hardware y/o software. Incluso los propios gestores pueden necesitar acceder a los recursos compartidos gestionados por otros procesos de tal forma que algunos procesos son clientes y servidores simultáneamente.

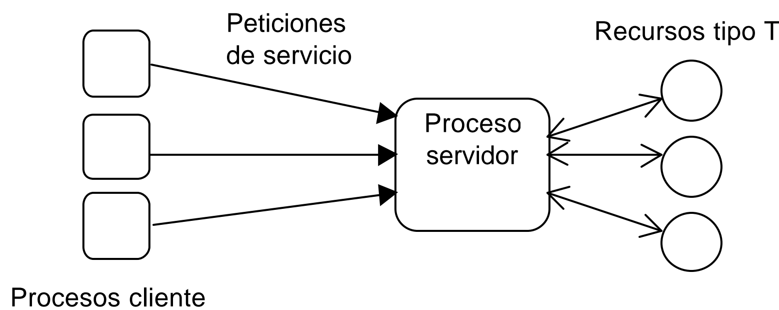


Figura 3.1. Modelo de procesos.

Los servidores se encargan de organizar eficientemente el acceso a los recursos que gestionan. Cada servidor proporciona un conjunto de operaciones, que deberán ser utilizadas por los clientes para actuar sobre los recursos.

Los clientes emiten peticiones a los servidores cuando necesiten acceder a alguno de sus recursos. Si la petición es válida, el servidor efectúa la operación solicitada y envía el resultado de su realización al cliente.

Este modelo proporciona una aproximación de propósito general a la compartición de recursos e información en sistemas distribuidos. Se puede implantar en una amplia variedad de entornos hardware y software. Los computadores usados pueden ser de muchos tipos y tanto clientes como servidores pueden estar ejecutándose en la misma o distintas máquinas.

En esta visión tan simple los procesos servidores podrían verse como suministradores centralizados de los recursos que manejan, hecho no deseable en un sistema distribuido (problema de falta de escalabilidad). Por esta razón, es necesario desligar los conceptos de servicio proporcionado a un cliente y servidor que lo proporciona. Un **servicio** es una entidad abstracta que puede ser proporcionado por varios procesos servidores que se están ejecutando en distintas máquinas y que cooperan vía una red de comunicación.

Sin embargo, por motivos de eficiencia, hay algunos recursos que deben permanecer siempre locales a cada máquina: memoria RAM, procesador central y la interfaz de red local.

3.2.2 Un modelo intermedio

El sistema operativo Plan 9 [PPD+90, PPT+93] va un poco más allá en el tratamiento de los recursos. En Plan 9, todos los **recursos se muestran como sistemas de ficheros**: se organizan en un árbol jerárquico y se accede a su contenido con operaciones de lectura y escritura (*read*, *write*). Esto no quiere decir que sean repositorios de ficheros permanentes en disco, sino que la interfaz para acceder a ellos es orientada a fichero.

A este nivel de abstracción, los ficheros de Plan 9 son similares a los objetos, excepto que los ficheros ya tienen incorporados métodos de nombrado, acceso y protección, que sería necesario crear para los objetos. Sobre el modelo de objetos se hablará en el apartado siguiente.

Por razones de eficiencia y de extrema dificultad de implantación, existen, sin embargo, determinadas operaciones en Plan 9 que no se han hecho corresponder con operaciones de entrada/salida en ficheros: la creación de procesos, la introducción y manipulación del espacio de nombres de red en el sistema de ficheros y la memoria compartida.

3.2.3 Modelo basado en objetos

El modelo basado en objetos afronta la construcción del sistema identificando las **abstracciones clave** del dominio del problema. Dichas entidades son vistas como agentes autónomos que colaboran para **llevar a cabo un comportamiento de nivel superior**.

El modelo basado en objetos pone énfasis en la identificación y caracterización de los componentes del sistema, que serán modelados con objetos. Informalmente, se puede definir un **objeto** como una colección formada por una serie de datos y un conjunto de operaciones definidas sobre ellos, de tal manera que se garantiza la ocultación de los detalles internos.

Para alterar el estado de un objeto, se tendrá que invocar la operación apropiada, de tal forma que el conjunto de las operaciones definidas para un objeto definirá colectivamente su comportamiento. El mecanismo de encapsulación de los objetos se encarga de que esto sea así.

También se pueden utilizar los **mecanismos de herencia y polimorfismo** como herramientas adicionales de estructuración para el diseño y construcción de sistemas distribuidos. La aplicación de conceptos de orientación a objetos tal cual para la construcción de sistemas distribuidos es denominada en [BG96] **aproximación aplicativa**. Un ejemplo clásico del uso de estos mecanismos es el sistema operativo Choices [CIR+93].

El modelo basado en objetos no difiere del tradicional paradigma de OO en el que cada entidad es un objeto con una interfaz de mensajes que proporcionan sus operaciones. En el modelo basado en objetos para los sistemas distribuidos, cada recurso compartido se representa como un objeto. Los objetos son identificados de forma unívoca y se pueden mover por la red sin cambiar su identidad. Cuando un proceso requiere el acceso a un recurso, envía un mensaje al objeto correspondiente. Este mensaje es despachado al método adecuado que resuelve la petición y envía un mensaje de respuesta al proceso adecuado.

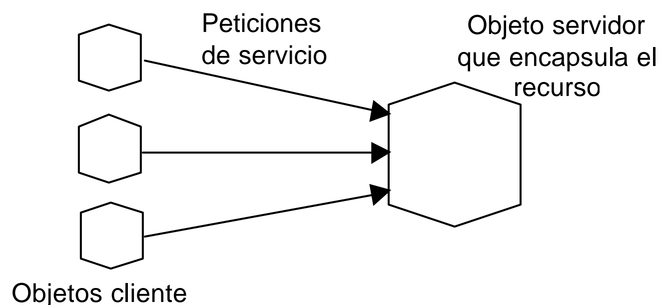


Figura 3.2. Modelo basado en objetos.

El modelo basado en objetos es **simple y flexible**. Ofrece una **visión uniforme** de todos los recursos. Como en los modelos descritos anteriormente, los objetos pueden actuar como clientes y servidores. Sin embargo, en el modelo cliente/servidor, el esquema de nombrado de los recursos dependía del gestor mientras que aquí todos los recursos se referencian de manera uniforme.

3.3 Modelo de objetos

Las características del modelo de objetos a implantar en un sistema de distribución de objetos no deben diferir de las que son consideradas de manera tácita como las más representativas de la OO, y que fundamentalmente se encuentran recogidas en la metodología de Booch [Boo94].

3.3.1 Características del modelo de objetos de Booch

Propiedades fundamentales:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía de herencia y de agregación

Propiedades secundarias:

- Tipos
- Concurrencia
- Persistencia
- Distribución

3.3.1.1 Abstracción y encapsulamiento. Clases

Estas propiedades suelen describirse como un conjunto. La **abstracción** se define en [Boo94] como *"Una abstracción denota las características coincidentes de un objeto*[1] *que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador"*. Podría resumirse indicando que un objeto presenta una determinada interfaz que muestra su comportamiento esencial. Este comportamiento se presenta mediante una serie de métodos u operaciones que se pueden invocar sobre el objeto.

Esta propiedad se suele combinar con la de **encapsulamiento**, que establece el principio de ocultación de información: *"el encapsulamiento es el proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento, sirve para separar la interfaz contractual de una abstracción de su implementación"*. La implementación de un objeto (la implementación de sus métodos y sus estructuras internas o estado interno) debe estar totalmente encapsulada en el mismo y separada de su interfaz. La única manera de utilizar un objeto es mediante los métodos de su interfaz. No se puede acceder directamente a la implementación de un objeto.

[1] Se utilizarán los nombres objeto e instancia indistintamente

También puede considerarse que el objeto encapsula en general toda su semántica, incluyendo las propiedades implícitas que puedan existir en el modelo de objetos, como la concurrencia, que se describen más adelante.

Como unidad de abstracción y representación del encapsulamiento, la mayoría de los sistemas OO utilizan el concepto de **clase**. Todos los objetos que tienen las mismas características se agrupan en una misma clase. La clase describe las características que tendrán todos los objetos de la misma: la representación del estado interno y su interfaz. Así, cuando se crea un nuevo objeto, se crea a partir de una clase, de la que toma su estructura.

3.3.1.2 Modularidad

La **modularidad**[2] permite fragmentar un problema complejo en una serie de conjuntos de objetos o módulos, que interactúan con otros módulos. En cierta manera, representa un nivel superior al nivel de encapsulamiento, aplicado a un conjunto de objetos.

Jerarquía. La relación "es-un" (herencia). La relación "es-parte-de" (agregación)

La **jerarquía** es "una clasificación u ordenación de abstracciones". Permite comprender mejor un problema agrupando de manera jerárquica las abstracciones (objetos) en función de sus propiedades o elementos comunes, descomponiendo en niveles inferiores las propiedades diferentes o componentes más elementales.

Las dos jerarquías más importantes son las representadas por las relaciones "**es-un**" y "**es-parte-de**".

Herencia. La relación "es-un"

Es una relación que se establece entre las diferentes clases de un sistema. Esta relación indica que una clase (**subclase**[3]) comparte la estructura de comportamiento (las propiedades) definidas en otra clase (**superclase**[4]). La clase "hereda[5]" las propiedades de la superclase. Se puede formar de esta manera una jerarquía de clases[6]. Un objeto de una clase también "es-un" objeto de la superclase (por ejemplo, un coche es un vehículo). En el caso en que una clase pueda heredar de más de una superclase se habla de **"herencia múltiple"** por contraposición al caso de **"herencia simple"**.

Agregación. La relación todo/parte ("es-parte-de")

Las jerarquías "es-parte-de" describen relaciones de agregación entre objetos para formar otros objetos de un nivel superior. Permiten definir un objeto en términos de agregación de sus partes, de objetos componentes más elementales (chasis, motor y carrocería son parte de un coche).

3.3.1.3 Tipos y polimorfismo

"Los tipos son la puesta en vigor de la clase de los objetos, de manera que los objetos de tipos distintos no pueden intercambiarse, o, como mucho, pueden

---

[2] Un excelente tratamiento sobre la modularidad y la orientación a objetos en general se encuentra en [Mey88, Mey97].

[3] O clase hija.

[4] O clase padre.

[5] O "deriva de".

[6] Las clases situadas por encima de una clase en la línea de herencia son los ancestros, antepasados o ascendientes. Las que derivan de ella son los descendientes.

intercambiarse sólo de formas muy restringidas". En realidad un **tipo** denota simplemente una estructura común de comportamiento de un grupo de objetos. Normalmente se identifica el concepto de tipo con el de clase[7], haciendo que la clase sea la única manera de definir el comportamiento común de los objetos. De esta manera la jerarquía de tipos se funde con la jerarquía de clases. Así un objeto de un subtipo (subclase) determinado puede utilizarse en cualquier lugar en el que se espera un objeto de sus tipos ancestros (clases ancestras).

La existencia de tipos permite aplicar las normas de comprobación de tipos a los programas, ayudando a la detección de un número de errores mayor en el desarrollo de un programa. La **comprobación de tipos** permite controlar que las operaciones que se invocan sobre un objeto forman parte de las que realmente tiene. En la comprobación de tipos puede haber varios niveles de comprobación: comprobación estricta en la compilación de un programa (**comprobación estática**), sólo comprobación en tiempo de ejecución (**comprobación dinámica**) o una estrategia híbrida.

Enlace estático y dinámico. Polimorfismo

El enlace hace referencia al momento en el que un nombre (o referencia) se asocia con su tipo. En el **enlace estático** esto se realiza en tiempo de compilación. De esta manera la invocación de una operación sobre un objeto siempre activa la misma implementación de la operación.

En el **enlace dinámico**, el tipo del objeto se determina en tiempo de ejecución y el sistema determinará cuál es la implementación de la operación que se invocará en función de cual sea el objeto utilizado. Esto normalmente se denomina **polimorfismo** puesto que podemos aplicar el mismo nombre de operación a un objeto y en función de cual sea este objeto concreto el sistema elegirá la implementación adecuada de la operación.

Este mecanismo de enlace dinámico permite crear programas generales en los que el tipo de objeto sobre el que operen sea determinado en tiempo de ejecución. Por ejemplo un programa gráfico que simplemente indique a un objeto que realice la operación de dibujar. En tiempo de ejecución se enlaza dinámicamente con la implementación de la operación "dibujar" correspondiente al objeto concreto en tiempo de ejecución que se trate: un círculo, un rectángulo, etc.

El polimorfismo se suele considerar como una propiedad fundamental del modelo de objetos. Combinado con la herencia ofrece un compromiso entre la comprobación de tipos en tiempo de ejecución y la flexibilidad del enlace dinámico. De esta manera se puede usar el polimorfismo en un árbol de herencia compartiendo el mismo nombre de operación para las superclase y sus subclases, pero restringiendo la utilización de las operaciones a las que se pueden aplicar sobre el objeto (las de su árbol de herencia), siempre que pertenezca a una de las clases de la jerarquía.

3.3.1.4 Concurrencia

"La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo". La mayoría de los lenguajes OO están pensados para sistemas tradicionales con conceptos como fichero y proceso. Esto hace tender a que los objetos se consideren entidades "teledirigidas" por el flujo de ejecución secuencial del programa. Sin embargo, entre las propiedades de un objeto está la de la **concurrencia**, es decir, la

[7] En este trabajo se asociará siempre el concepto de tipo con el de clase, salvo indicación explícita de lo contrario.

posibilidad de tener actividad independientemente del resto de los objetos. Esto permite describir sistemas de una manera más cercana a la realidad, en la que los objetos en muchos casos tienen funcionamiento independiente.

3.3.1.5 Persistencia

"Es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado)".

Para que los sistemas OO sean más cercanos a la realidad, es necesaria la **persistencia**, es decir, un objeto no tiene por qué dejar de existir por el simple hecho de la finalización del programa que lo creó. El objeto representa una abstracción que debe permanecer como tal hasta que ya no sea necesaria su existencia dentro del sistema. Hay que hacer notar que no sólo los datos de un objeto deben persistir, la clase a la que pertenece también debe hacerlo para permitir la utilización del objeto con toda su semántica.

3.3.1.6 Distribución

Aunque no denominada como tal, la propiedad de la **distribución** es referida con el término de **persistencia en el espacio**. Un objeto puede moverse (en el espacio) por los distintos ordenadores que conformen el sistema de computación, manteniendo todas sus propiedades como en el caso de la persistencia. Es decir, los **objetos son distribuidos**.

3.3.2 Características adicionales necesarias

Como se verá más adelante, es necesario **completar el conjunto de características** del modelo de objetos recogidas en el modelo de Booch para un sistema de distribución de objetos: relaciones generales de asociación, identidad única de los objetos, excepciones y seguridad.

3.3.2.1 Relaciones generales de asociación

En un sistema, además de las relaciones "es-un" y "es-parte-de", existen más relaciones entre objetos. Es conveniente conocer qué objetos están relacionados con otros. Esta información puede ser útil cuando se tienen en cuenta aspectos de distribución de los objetos. Por ejemplo, al mover un objeto a otro computador, es interesante mover también los objetos que están relacionados, puesto que es muy probable que se intercambien muchos mensajes entre sí. Con la relación "es-parte-de" se pueden conocer los objetos integrantes de otro objeto y moverlos a la vez. Sin embargo, en el caso de un libro y la estantería donde está situado, la relación que existe entre ellos no puede modelarse como "es-parte-de". Si no existe otro tipo de relación en el modelo, el sistema no tendría conocimiento de esta relación y no podría explotarlo al igual que lo puede hacer con la relación "es-parte-de".

De hecho, relaciones adicionales a las de agregación y herencia son utilizadas por las metodologías. La propia metodología de Booch dispone de relaciones de uso entre objetos.

Como compromiso, se añade un tercer tipo de relación al modelo de objetos: la **relación de asociación** "asociado-con", que represente en general la existencia de otras relaciones diferentes a las de herencia o agregación entre objetos.

3.3.2.2 Identidad única de objetos

Al igual que en el mundo real, todos los objetos tienen una identidad que los diferencia del resto de los objetos. En un sistema de gestión de objetos tiene que existir un mecanismo para que el sistema distinga un objeto de otros, al objeto de trabajar con el mismo. Al usarse sólo OO lo único que el sistema hará con un objeto es invocar uno de sus métodos. La manera de hacer esto es mediante un identificador que tenga cada objeto.

En lenguajes como C++, se utiliza la posición de memoria que ocupa el objeto para acceder al mismo. Sin embargo, en un sistema global, con necesidades de persistencia, distribución, etc. esto no es válido pues un objeto puede cambiar de computador. Es necesario utilizar un identificador global que identifique de manera unívoca cada objeto dentro del sistema hasta que el objeto desaparezca. Este identificador único será la única manera de acceder al objeto dentro del sistema. Como se verá, la utilización de un **identificador único** para los objetos facilita la realización de propiedades como la persistencia, distribución, etc. de manera transparente.

3.3.2.3 Excepciones

Las excepciones son un concepto que está orientado fundamentalmente a la realización de programas robustos mediante el manejo estructurado de los errores. Básicamente, una **excepción** es un evento que se produce cuando se cumplen ciertas condiciones (normalmente condiciones de error). En estos casos se "lanza una excepción". La excepción será "atrapada" por un manejador de excepciones que realizará el tratamiento adecuado a la misma.

Las excepciones son un concepto importante con soporte en lenguajes populares como C++ y Eiffel. Parece adecuado, pues, introducirlo en el modelo de objetos del sistema para promover el desarrollo de programas robustos.

3.3.2.4 Seguridad

Aunque no sea de gran importancia en las fases de diseño y análisis de aplicaciones, en un sistema de computación real completo la seguridad es muy importante. Debe existir un mecanismo que permita la protección de los recursos del sistema, en este caso de los objetos.

En el caso de un sistema de distribución de objetos, el objetivo de la protección son los objetos, más concretamente la utilización de los métodos de un objeto por parte de otros objetos. Se trata de posibilitar que sólo ciertos objetos puedan invocar métodos de otros objetos y en el caso de cada objeto qué métodos concretos puede invocar. Por ejemplo, un objeto estudiante podría invocar el método leerNota de un objeto acta, pero no el método ponerNota. Un objeto profesor podría invocar ambas operaciones. Es importante que al igual que en el resto de los conceptos del sistema, este mecanismo sea uniforme e igual para todos los objetos.

3.4 Aplicación de las tecnologías de objetos a la construcción de sistemas distribuidos

En la actualidad, ingeniería del software y metodologías orientadas a objetos son casi sinónimos. Los objetos están presentes en todas las fases del desarrollo de software: análisis, diseño e implementación. Existen gran cantidad de libros y artículos que

hablan de los beneficios de la utilización de las tecnologías de objetos en el desarrollo de software, de los que destacan [Boo94, Mey97].

Debido a su tamaño y complejidad, los sistemas operativos han sufrido históricamente de problemas de escalabilidad, mantenibilidad y extensibilidad. Los sistemas distribuidos han venido a exacerbar dichos problemas. La utilización de tecnologías orientadas a objetos se propone como solución a estos problemas de ingeniería [MR93].

Cuando confluyen construcción de sistemas distribuidos y desarrollo de software con tecnologías de objetos, surgen inmediatamente una serie de preguntas ¿Dónde encajan los objetos en la construcción de sistemas distribuidos (entiéndase aquí sistema distribuido en su sentido más amplio)? ¿Es posible mantener los beneficios de la encapsulación, herencia y polimorfismo en un sistema distribuido construido con tecnologías de objetos?

Básicamente, la aplicación de las tecnologías de objetos a la construcción de sistemas distribuidos se puede realizar a dos niveles distintos [Kra93, CJM+91]: estructurando e implementando el sistema como un conjunto de objetos y proporcionando soporte para los objetos.

Con el objetivo de conseguir los mayores beneficios en la utilización de las tecnologías de objetos será preciso alentar la construcción de sistemas distribuidos aplicando los dos enfoques.

3.4.1 Diseño del sistema distribuido como un marco orientado a objetos

Se trata en este caso de **aplicar los principios de diseño e implantación del paradigma de la orientación a objetos** al propio sistema distribuido, de forma que quede organizado como un marco orientado a objetos. El sistema distribuido se diseña como una jerarquía de clases, cada una de las cuales ofrece determinada funcionalidad.

Los beneficios de esta aproximación son dobles: por un lado, el sistema tiene una estructura modular, en la cual cada uno de sus componentes puede ser identificado y reutilizado; por otro lado, con el uso de mecanismos como la herencia, el sistema puede ser configurado para unas necesidades específicas o para cumplir determinadas restricciones.

Los objetos se presentan como una herramienta muy útil para estructurar los sistemas distribuidos por varias razones [Kra93]:

- Los objetos proporcionan modularidad y encapsulación, dado que separan claramente la interfaz de la implementación.
- La comunicación cuenta con un mecanismo uniforme de alto nivel, la **invocación de métodos** o **paso de mensajes** entre objetos, que es bien conocido y para el que existen técnicas para conseguir implementaciones eficientes.
- Los objetos proporcionan un medio adecuado para la compartición de información. La ventaja respecto a los ficheros compartidos es una semántica de más alto nivel y una mejor protección ante errores de programación.
- Los sistemas estructurados en términos de objetos evolucionan fácilmente (más fáciles de extender y mantener), por reemplazo de sus partes. Además los subsistemas y objetos individuales podrán ser reutilizados con facilidad y

adaptados a nuevas condiciones de operación o entorno vía el mecanismo de herencia.

El sistema distribuido se estructura internamente como un conjunto de objetos que implementan la funcionalidad del sistema. Además, en tiempo de ejecución, se instanciarán objetos reales de las distintas clases en que se oferta la funcionalidad del sistema, con la semántica que ello conlleva.

3.4.2 Diseño del sistema distribuido como soporte de objetos

En este caso, el **sistema proporciona soporte para los objetos**, es decir, el objeto es el elemento básico del sistema, en claro contraste con los habituales proceso y fichero. Además, el sistema proporciona un entorno que soporta la definición dinámica de jerarquías de clases y herencia, tanto para los programas de sistemas como para las aplicaciones. Sus beneficios son, en general (y principalmente para los desarrolladores de aplicaciones), los mismos que los que proporcionan las tecnologías de objetos. El soporte para los objetos se puede dar, a su vez, a diferentes niveles:

- Incluyendo las funciones de gestión de objetos en el propio sistema operativo.
- Proporcionando una capa de gestión de objetos por encima de un sistema operativo existente.
- Implementando la capa de objetos como una capa de soporte en tiempo de ejecución (run-time) para lenguajes de programación específicos (lenguajes orientados a objetos).
- El soporte viene dado por el hardware (o máquina abstracta) subyacente.

3.5 Cuestiones de diseño del soporte de objetos en el sistema operativo distribuido

Una vez decidido que los objetos (por extensión, las tecnologías orientadas a objetos) son adecuados para construir sistemas distribuidos (de nuevo, por extensión, sistemas operativos distribuidos) y que es necesario que estos proporcionen soporte para los objetos, el diseñador debe tomar en consideración la **estructura de los objetos** que van a ser soportados por el sistema distribuido: actividad interna, granularidad, visibilidad, volatilidad/persistencia, movilidad, replicación y composición.

Es necesario, en este momento, integrar conceptos como la relación entre procesos y objetos, el soporte de réplicas por parte del sistema, etc. Se trata, en definitiva, de unificar conceptos de paralelismo y distribución en general con conceptos de orientación a objetos. Dicha unificación recibe en [BG96] el nombre de **aproximación integradora**.

En el capítulo 10 se identifican y describen las opciones adoptadas para cada una de las cuestiones de diseño presentadas a continuación.

3.5.1 Actividad interna

La relación entre los procesos y los objetos de un sistema de distribución de objetos caracteriza la actividad interna de los objetos. Los procesos pueden estar separados y relacionados de manera temporal con los objetos que invocan, o bien, pueden estar asociados de manera permanente a los objetos en los que se ejecutan. Estas dos aproximaciones se corresponden, respectivamente, con el **modelo de objetos pasivo** y el **modelo de objetos activo**.

3.5.1.1 Modelo de objetos pasivo

El modelo de objetos pasivo se basa en ofrecer una clase de objetos especiales, representativa de la computación, equivalente al concepto de proceso en los sistemas operativos tradicionales. Los demás objetos son meros contenedores de datos y los métodos que los manipulan.

Un proceso no está restringido a un solo objeto, sino que puede ejecutarse en varios objetos durante su existencia para satisfacer una acción. Cuando un proceso realiza una invocación en otro objeto, su ejecución en el objeto actual se suspende. Conceptualmente, el proceso se asocia al espacio de direcciones del segundo objeto, donde ejecuta la operación apropiada. Cuando se completa dicha operación, el proceso vuelve al primer objeto, donde reanuda su ejecución de la operación original.

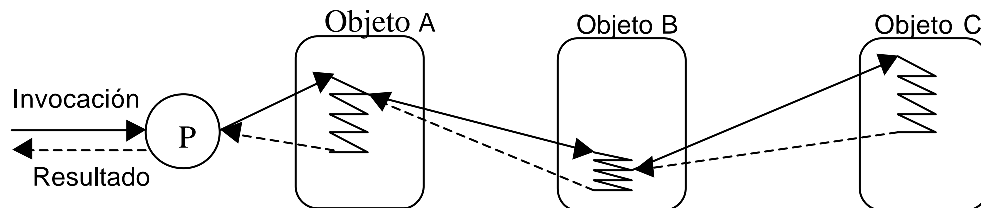


Figura 3.3. Realización de una acción en el modelo de objetos pasivo.

Una ventaja de la utilización del modelo de objetos pasivos es que virtualmente no hay restricción en el número de procesos asociados a un objeto. Como inconvenientes están las dificultades y costes de asociar y desasociar procesos a los espacios de direcciones de los objetos en los que se ejecutan.

Adicionalmente, el modelo de objetos pasivo presenta una gran dificultad para la distribución de los objetos. Sería necesario transformar los procesos en procesos distribuidos, construidos en base a estructuras de computación distribuidas, de tal forma que en todo momento sea factible conocer la secuencia de ejecución de métodos, tanto en una máquina como en varias.

3.5.1.2 Modelo de objetos activo

El modelo de objetos activo considera los objetos como entidades autónomas, que encapsulan datos, métodos y computación. De alguna manera, es como si los procesos formasen parte de los objetos. Con esta aproximación, múltiples actividades independientes pueden estar involucradas en la realización de una acción. La interacción entre objetos se realiza mediante **paso de mensajes** que contienen todos los datos necesarios para la invocación.

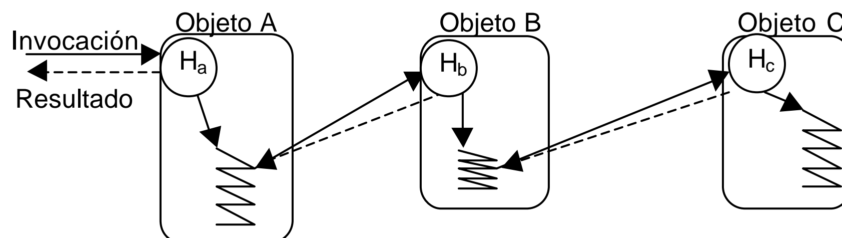


Figura 3.4. Realización de una acción en el modelo de objetos activo.

El modelo de objetos activo permite que el sistema de gestión de objetos ofrezca una sola entidad (el objeto) con un contenido semántico superior al modelo pasivo. De esta

forma, ante la realización de invocaciones concurrentes a sus métodos, un objeto puede decidir aceptar o retrasar la ejecución de las mismas de acuerdo a su estado interno.

Entre otras cosas, el número máximo de procesos que un objeto puede albergar deberá ser tenido en cuenta para cada invocación que llega, de manera que todas las invocaciones a un objeto en el que ya existe el máximo de procesos deberán ser rechazadas o encoladas para un tratamiento posterior.

El principal inconveniente que se le achaca a este modelo es la eficiencia, debido a la sobrecarga que produce el incluir en la representación en tiempo de ejecución del objeto las estructuras que permiten asociar a cada objeto su computación.

Las ventajas que proporciona, son varias. En primer lugar, ofrece una única abstracción, el objeto, para modelar los sistemas. Estos objetos son autocontenidos, es decir, incluyen datos, métodos y computación y, por tanto, son quienes la gestionan. Además, dado que los objetos realizan sus acciones ante la llegada de mensajes, el propio mecanismo de paso de mensajes sirve para activar métodos y sincronizar ejecuciones.

El modelo de objetos activo es muy adecuado para la distribución de objetos. Al proporcionar una visión única del objeto, la computación se hace transparente al mecanismo de la distribución. Así, la migración de un objeto comprende también la migración de su computación. Por su parte, el mecanismo de paso de mensajes tiene que ser incrementado para que estos puedan enviarse a través de una red.

3.5.2 Granularidad de los objetos

El tamaño, sobrecarga y cantidad de procesamiento realizada por un objeto caracterizan su **granularidad**. Este aspecto es muy importante, dado que está relacionado directamente con la usabilidad, rendimiento y homogeneidad en la utilización de los objetos como tecnología de construcción de sistemas distribuidos. Se distinguen tres grados en la granularidad de los objetos: grano grueso, grano medio y grano fino.

3.5.2.1 Objetos de grano grueso

Los objetos de **grano grueso** se caracterizan por un gran tamaño, un número relativamente grande de instrucciones a ejecutar para servir una invocación y un número relativamente reducido de interacciones con otros objetos. Este tipo de objetos se asocia inmediatamente con los procesos servidores de recursos del modelo cliente/servidor tradicional (ver apartado 3.2.1 de este mismo capítulo).

La utilización de un esquema puro de objetos de grano grueso ofrece como ventaja la simplicidad, pero existen ciertos inconvenientes cuando se desea implantar en un sistema de distribución de objetos: en primer lugar, el control y la protección de los datos están al nivel de los objetos de grano grueso, lo cual restringe la flexibilidad del sistema y la cantidad de concurrencia entre objetos que puede proporcionar. En segundo lugar, el sistema no va a poder proporcionar un modelo de datos homogéneo. Las grandes entidades de datos se representarán con objetos, pero las pequeñas entidades, como listas enlazadas o enteros, se tendrán que representar con abstracciones de datos convencionales de lenguajes de programación.

3.5.2.2 Objetos de grano medio

Para proporcionar un nivel de control más fino sobre los datos, el sistema puede soportar tanto objetos de grano grueso como objetos de **grano medio**. Los objetos de

grano medio son más fáciles (baratos) de crear y mantener, dado que son más pequeños en tamaño y menos ambiciosos en cuanto a su funcionalidad, que los objetos de grano grueso. Ejemplos típicos de estos objetos son estructuras de datos como una lista enlazada y una cola. Como inconvenientes de los objetos de grano medio están la sobrecarga adicional derivada del mayor número de objetos a ser gestionados por el sistema y que su existencia no proporciona todavía un modelo de datos homogéneo (entidades como los enteros siguen siendo representados con abstracciones de los lenguajes de programación).

3.5.2.3 Objetos de grano fino

Finalmente, para proporcionar un nivel aún más fino de control sobre los datos, el sistema puede soportar objetos de grano grueso, de grano medio y de **grano fino**. Los objetos de grano fino se caracterizan por su pequeño tamaño, un reducido número de instrucciones a ejecutar y un gran número de interacciones con otros objetos. Ejemplos de este tipo de objetos se encuentran en tipos de datos proporcionados por los lenguajes de programación, como los enteros, reales, lógicos, etc. El mayor problema de los sistemas que utilizan objetos de grano fino es la pérdida de rendimiento derivada de la gestión de un gran número de objetos. Sin embargo, esta aproximación proporciona un entorno homogéneo, donde toda entidad es un objeto, independientemente de su tamaño.

La experiencia ha mostrado que los programadores tienden a utilizar objetos pequeños (de unos cientos de bytes), mientras que los mecanismos básicos del sistema (comunicación, compartición, etc.) son más eficientes para objetos de grano grueso [Kra93]. Esto último suele ser así en diferentes sistemas distribuidos que proporcionan soporte de objetos (ver capítulo 4), en los que estos juegan un papel similar al de los procesos en los sistemas operativos tradicionales.

3.5.3 Visibilidad: objetos privados/objetos compartidos

Los **objetos privados** son visibles únicamente en el ámbito de los objetos que los crean. Juegan habitualmente el papel de atributos agregados de otros objetos o bien existen de manera transitoria como objetos locales de un método.

Los **objetos compartidos** ofrecen un medio de comunicación uniforme entre objetos, aunque implican la necesidad de mecanismos de protección y sincronización, dado que van a ser invocados por varios objetos incluso de manera simultánea. Juegan el papel de los procesos servidores del modelo de procesos descrito al principio del capítulo.

Habitualmente coexistirán objetos privados y compartidos.

3.5.4 Objetos volátiles/objetos persistentes

Los **objetos volátiles** existen durante el tiempo que el sistema esté funcionando. Juegan el papel de las variables de los sistemas tradicionales.

Los sistemas tradicionales adolecen del problema de utilizar abstracciones distintas en función del tiempo que se desee que perduren los datos que manejan: para los datos en memoria volátil se utilizan variables, de distintos tipos, y para los datos en memoria persistente, se utilizan los ficheros.

Los **objetos persistentes** unifican ambas visiones, ofreciendo una abstracción única e independiente de la localización instantánea de un objeto. Como inconveniente de la

introducción de objetos persistentes distribuidos destaca la dificultad a la que pueden llevar la tarea de la recolección de basura.

A la vez que un objeto persistente reside en memoria secundaria puede existir alguna copia volátil del mismo en memoria principal. Las operaciones sobre el objeto modificarán la copia volátil, hasta que se tome la decisión de actualizar la copia persistente. En el caso de que se realizase una invocación a un objeto que únicamente dispusiese de la copia persistente, se crearía automáticamente una copia volátil para atender la invocación. El sistema de persistencia decidirá cuándo prescindir de copias volátiles de objetos que se encuentran actualizados en el almacenamiento de persistencia.

3.5.5 Movilidad: objetos fijos/objetos móviles

La migración de procesos en sistemas distribuidos tradicionales es muy útil para mejorar el reparto de carga de los diferentes computadores con el fin de mejorar el rendimiento global y para ciertas restricciones de administración o seguridad.

Si se utiliza como unidad de migración el objeto, se obtienen inmediatamente **objetos móviles**. En contraposición, los **objetos fijos** no se moverán del nodo en el que fueron creados durante toda su existencia, pudiendo, en principio, comunicarse igualmente con cualquier otro objeto del sistema distribuido.

Los objetos móviles migrarán entre los diferentes nodos del sistema distribuido por razones similares a las que se tomaban para la migración de procesos en sistemas distribuidos tradicionales [Gos91]. También se tendrán que tomar decisiones sobre los objetos susceptibles de ser migrados, si la migración se plantea individual o colectivamente, etc.

El inconveniente principal de los objetos móviles es que hacen la localización de objetos más compleja.

3.5.6 Replicación: objetos de una copia/ objetos replicados

Un esquema de **objetos de una copia** proporciona una identidad única y diferente a todos los objetos del sistema distribuido. La no-disponibilidad de un nodo del sistema distribuido podría hace inaccesibles todos los objetos de este tipo que residían en él en el momento en que se produjo el problema.

Un esquema de **replicación de objetos** permite la existencia de múltiples copias de un objeto en diferentes nodos del sistema distribuido, de manera que es tolerable el fallo de algún nodo mientras todavía se proporciona una funcionalidad casi total. El fallo de un nodo cualquiera solo resultaría en la no-disponibilidad de las réplicas que residen en el mismo, con el problema adicional de la merma en el rendimiento del sistema.

Como inconvenientes de la replicación de objetos destacan la dificultad del mantenimiento de información consistente entre réplicas del mismo objeto y la sincronización de actividades de múltiples clientes.

La forma más sencilla de introducir algún aspecto de replicación de objetos en un sistema distribuido es aplicándola únicamente a **objetos inmutables**. Los objetos inmutables son aquellos que no pueden ser modificados una vez que han sido creados. La replicación de este tipo de objetos se simplifica notablemente, dado que no es necesario mantener un estado consistente ni sincronizar las operaciones de acceso al mismo. Como inconveniente aparece su uso limitado, dado que sólo es aplicable a objetos no modificables.

En el caso de que se opte por un esquema de objetos replicados, el modelo de objetos implantado por el sistema distribuido debería ocultar este hecho tras una interfaz uniforme, de tal forma que la réplica concreta que realmente vaya a servir una petición de un cliente sea irrelevante para éste.

3.5.7 ¿Qué es, entonces, un objeto distribuido?

Como se ha comentado anteriormente, un objeto representa una unidad independiente de ejecución que encapsula datos, procedimientos y, posiblemente, recursos privados (actividad) para el procesamiento de las peticiones. Por lo tanto, una opción natural es considerar el **objeto como unidad de distribución y posible replicación**.

Adicionalmente, la autocontención de los objetos (datos + métodos + posible actividad interna) facilita la posibilidad de su migración. Además, el paso de mensajes no solo asegura la separación entre los servicios ofertados por un objeto y su representación interna, sino que también proporciona independencia de su localización física. Por tanto, el paso de mensajes aglutinará tanto la invocación local como remota (los objetos emisor y receptor residen en el mismo o diferentes nodos), así como la posible no disponibilidad de un objeto a pesar de existir.

Para considerar a un **objeto distribuido** como tal, el sistema operativo tiene que dotarlo de las propiedades siguientes:

- Que pueda ser **invocado tanto local como remotamente**. El objeto podrá recibir invocaciones por parte de otros objetos con independencia de sus respectivas ubicaciones dentro del sistema distribuido.
- Que pueda **migrar** entre diferentes nodos del sistema distribuido. De acuerdo con las políticas de migración de objetos que se establezcan, el objeto puede modificar su ubicación en el sistema distribuido sin que ello afecte a la computación estuviese en curso relativa al objeto que.

3.6 Problemas de la gestión de objetos

La gestión de los objetos de un sistema distribuido introduce un conjunto de problemas nuevos y matiza alguno de los ya existentes en el diseño y construcción de sistemas distribuidos. A continuación se describen los más relevantes.

3.6.1 Nombrado

En un sistema distribuido, los **nombres** se utilizan para hacer referencia a los diferentes recursos, como servicios, usuarios, estaciones de trabajo, etc. En un sistema distribuido orientado a objetos la única entidad que existe es el objeto, y serán los objetos los referenciados mediante los nombres.

El **nombrado de los objetos** es una función básica de todo sistema de gestión de objetos, y puede ser llevada a cabo a diferentes niveles. Nos encontramos habitualmente con nombres simbólicos, de alto nivel, y nombres internos, de bajo nivel.

Los **nombres internos**, también denominados **identificadores**, se eligen de manera que las operaciones que los manipulan se puedan realizar de manera eficiente. En algunos casos pueden venir acompañados de información de seguridad, que incluye derechos para realizar operaciones sobre objetos y que será difícil de falsificar. En otros casos, pueden venir acompañados de información que proporcione una idea de la

ubicación actual del objeto. Los nombres internos pueden ser, a su vez, locales (relativos a un contexto dado) o universales, que son únicos en el espacio y el tiempo. Todo objeto tiene siempre un nombre interno.

Los **nombres simbólicos** son utilizados para hacer referencia a los objetos de manera que sean sencillos de reconocer. Como ejemplo típico de nombres simbólicos están los nombres de ficheros que utilizan los usuarios de los sistemas operativos convencionales. Todo objeto puede tener cero o más nombres simbólicos asociados.

Los nombres simbólicos se asocian con los nombres internos utilizando lo que se denomina **servicio de nombrado**, que almacena una base de datos de relaciones entre ambos y que puede ser consultada y actualizada ante peticiones de los usuarios.

3.6.2 Localización

El problema de la localización de objetos consiste en **determinar la ubicación** actual (nodo o procesador) de un objeto a partir de su nombre. Si el nombre del objeto es puro, es decir, no contiene información de localización, el problema de la localización de un objeto de manera eficiente en un sistema grande es de gran dificultad. La difusión (*broadcast*) de mensajes para la localización debería ser utilizada como último recurso, dado que supone un gran coste en un sistema distribuido de gran tamaño. Existen técnicas que se basan en la localidad de acceso a los objetos (las referencias a objetos tienden a agruparse) y su permanencia (los objetos tienden a moverse con poca frecuencia) para optimizar la operación de localización.

3.6.3 Acceso

El acceso a un objeto (invocación de uno de sus métodos) involucra la siguiente secuencia de operaciones: localizar el objeto; comprobar la conformidad de los tipos (si no se hizo en tiempo de compilación); determinar el método a invocar (si es posible la invocación dinámica); finalmente, ejecutar la llamada. La ejecución del método puede terminar con éxito, devolviéndose un resultado, o bien fracasar, en cuyo caso sería importante que el sistema dispusiese de un modelo de excepciones.

3.6.4 Compartición y protección

La compartición de objetos se presenta como un modelo natural de comunicación entre objetos. La compartición secuencial, es decir, no concurrente, puede ser implementada con objetos persistentes. La compartición concurrente debe involucrar mecanismos para el control de la concurrencia, que podrían llegar al nivel de complejidad de las transacciones [HR83].

La protección debe ser adaptada al modo en que los objetos son utilizados, es decir, se debe definir en términos de los métodos que se permite invocar en lugar de los derechos de acceso tradicionales basados en permisos de lectura/escritura.

3.6.5 Persistencia

Uno de los problemas principales a resolver por el subsistema de persistencia es el de la diferencia de las referencias a objetos volátiles y persistentes, conocido como transformación de punteros (*pointer swizzling*), que se habrá de solucionar ofreciendo una abstracción única de referencia a objeto, independiente de su estado.

Otra cuestión importante relacionada con la persistencia es la recolección de basura. Se tiene que llegar a una solución de compromiso entre eficiencia y seguridad, lo que

lleva a utilizar métodos pesimistas: algo de "basura" puede llegar a no ser eliminada nunca.

3.7 Características de un sistema operativo distribuido orientado a objetos

En la bibliografía tradicional de sistemas operativos es difícil encontrar una definición como tal de sistema operativo. Los autores definen, en general, los sistemas operativos mostrando una relación de las funciones que debe realizar. Intentar dar aquí una definición de lo que es un sistema operativo distribuido orientado a objetos sería muy osado, por lo que se va a seguir la pauta marcada por otros autores. La definición vendrá dada como un compendio de las características y funciones que vamos a exigir a todo aquel software que quiera ser catalogado como un sistema operativo distribuido orientado a objetos.

Un **sistema operativo distribuido orientado a objetos** será aquel que, además de cumplir con la definición que para un sistema operativo distribuido aparece en el capítulo anterior, tiene las características siguientes:

- Está construido como un conjunto de objetos, o lo que es lo mismo, el sistema operativo está diseñado como un marco orientado a objetos, en el que cada uno de los objetos proporciona una funcionalidad específica del mismo.
- Proporciona soporte para objetos de nivel usuario. Los objetos no se circunscriben únicamente al diseño y la construcción del sistema operativo, sino que se convierten en el elemento fundamental para la construcción de aplicaciones. La unión de esta característica y la anterior concluyen que todo en el sistema operativo es un objeto.
- Proporciona un mecanismo uniforme para la comunicación entre objetos de aplicación y objetos del sistema y entre los propios objetos de aplicación, que se concreta en la invocación de métodos. La tradicional solicitud de servicios al sistema operativo se realiza por invocación de métodos de objetos que proporcionan su funcionalidad. De la misma manera, la interacción entre las aplicaciones se concreta en invocaciones de métodos.
- La invocación de métodos debe ser proporcionada de manera transparente e independiente de la ubicación de los objetos (local o remota), el estado de los objetos (volátil o persistente) y la función del objeto (sistema o aplicación). La transparencia en todos estos aspectos es fundamental, dado que se consigue el resultado de que el programador únicamente se tiene que preocupar del dominio del problema a resolver. En tiempo de ejecución, el sistema operativo se encargará de solventar todos los problemas que surgen debidos a las diferentes situaciones en que se pueden encontrar los objetos, sin necesidad de que el programador haya tenido que escribir código para su tratamiento.
- Proporciona mecanismos para la migración de objetos entre los distintos nodos del sistema distribuido. Todo sistema operativo debe hacer un uso racional y eficiente de los recursos. Las unidades de cómputo (procesadores o computadores) son unos de los recursos más críticos, de tal forma que no es admisible tener algunas ociosas mientras otras soportan una gran cantidad de tareas. El sistema operativo deberá equilibrar la carga de las diferentes unidades de cómputo con el fin de maximizar el rendimiento global. Así mismo, las

operaciones a realizar que requieren el envío de mensajes a través de la red sufren de un considerable retraso con respecto a las que se pueden resolver dentro de una misma unidad de cómputo. Es importante, entonces, que todas aquellas entidades involucradas en la realización de una operación estén lo más próximas entre sí que sea posible. En un sistema operativo distribuido orientado a objetos, la computación a realizar está asociada a los propios objetos, de tal manera que mover carga computacional de una unidad de cómputo a otra se traduce en mover (migrar) objetos.

En esta Tesis se va a utilizar el conjunto de características anterior para delimitar el alcance del sistema de distribución de objetos del sistema operativo orientado a objetos diseñado. Como características adicionales, pero no secundarias, exigiremos al sistema operativo que sea extensible (que sea sencillo introducir nuevos servicios) y flexible (adaptable).

3.8 Resumen

La construcción de un sistema distribuido no es trivial. Se distinguen básicamente dos modelos para afrontar su diseño y construcción: el modelo de procesos o cliente servidor, y el modelo basado en objetos.

El modelo de procesos utiliza la abstracción de proceso habitual de los sistemas operativos convencionales para construir el sistema. Todos los recursos disponibles para las aplicaciones de usuario son gestionados por procesos servidores, que definen el conjunto de operaciones permitidas. Por su parte, los procesos clientes, solicitan a los procesos servidores la utilización de los recursos. La comunicación entre clientes y servidores se realiza a través de alguna forma de paso de mensajes.

El modelo basado en objetos utiliza el concepto de objeto para abstraer las entidades que componen el sistema. Todos los recursos son presentados en forma de objetos, y las aplicaciones utilizan los recursos invocando los métodos que proporcionan los objetos que los encapsulan. Las características básicas que tendrán que presentar los objetos son: abstracción, encapsulamiento, modularidad y jerarquías de herencia y agregación.

Las tecnologías de objetos están extendiendo su ámbito de aplicación al diseño y construcción de sistemas distribuidos, con dos enfoques distintos: construyendo el sistema distribuido como un marco orientado a objetos y proporcionando soporte para los objetos. Ambos enfoques tienen sus ventajas e inconvenientes y no son excluyentes, de manera que parece lógico pensar en utilizar los dos para la construcción de un sistema distribuido orientado a objetos.

Una vez adoptados los objetos como abstracción sobre la que construir sistemas distribuidos, el diseñador debe tomar en consideración la estructura de los objetos que van a ser soportados: actividad interna, granularidad, visibilidad, volatilidad/persistencia, movilidad, replicación y composición.

Un sistema operativo distribuido orientado a objetos ofrecerá la funcionalidad de un sistema operativo distribuido con un diseño orientado a objetos y en el que el objeto se convierte en la abstracción básica y única. La comunicación entre aplicaciones y aplicaciones y sistema operativo se realizará a través de invocaciones a objetos que encapsulen la funcionalidad deseada. El sistema operativo proporcionará, de manera transparente, facilidades para la invocación remota y migración de objetos, como funcionalidad propia de su carácter distribuido y orientado a objetos.

CAPÍTULO 4 PANORÁMICA DE SISTEMAS DISTRIBUIDOS ORIENTADOS A OBJETOS

4.1 Introducción

En este capítulo se revisan diferentes sistemas operativos y "*middleware*" que comparten algunas de las características deseables para un sistema distribuido orientado a objetos. Se trata de detectar características comunes y estrategias que sean de utilidad para el diseño de una arquitectura para el sistema de distribución de objetos de un sistema operativo.

Los sistemas revisados intentan ser representativos de las diferentes tendencias actuales y son una selección de los sistemas examinados. Los aspectos relevantes de estos y otros sistemas para apartados específicos del sistema de distribución se examinarán posteriormente al tratar estos apartados.

4.2 DCE y DC++

DCE (*Distributed Computing Environment*) de *Open Software Foundation* [OSF92], ahora denominada *Open Group*, es un conjunto integrado de herramientas y servicios que soportan el desarrollo de aplicaciones distribuidas, de entre los que destacan las **llamadas a procedimientos remotos** (RPC, *Remote Procedure Call*), el **servicio de directorio de celda** (CDS, *Cell Directory Service*), los **servicios globales de directorio** (GDS, *Global Directory Services*), el **servicio de seguridad**, los **hilos** DCE, y el **servicio de ficheros distribuidos** (DFS, *Distributed File Service*).

DCE es una tecnología de tipo "*middleware*" o "habilitadora" que no tiene sentido de existir por sí misma, sino más bien como un extra de un sistema operativo. Funciona en diferentes tipos de computadores, sistemas operativos y redes, facilitando la portabilidad del software, al ocultar las particularidades del entorno en que se ejecuta.

4.2.1 Modelo de programación

El modelo de programación de DCE es el **modelo cliente/servidor**. Las dos facilidades que ofrece DCE y que no pueden considerarse como servicios son las llamadas a procedimientos remotos (RPC) y los hilos.

4.2.2 RPC

La **llamada a procedimiento remoto** (RPC, *Remote Procedure Call*) es la facilidad que hace posible a un programa cliente acceder a un servicio remoto invocando simplemente un procedimiento local. Es responsabilidad del sistema de RPC ocultar todos los detalles a los clientes y servidores: localizar el servidor correcto, construir y transportar los mensajes en ambas direcciones y realizar todas las conversiones de tipos necesarias entre el cliente y servidor, salvando las posibles diferentes arquitecturas en las que se ejecutan ambos.

Para que un servidor especifique los servicios que oferta a sus clientes, debe construir una interfaz con el **lenguaje de definición de interfaces** (IDL, *Interface Definition Language*). Toda interfaz tiene asignado un **identificador único** (UUID, *Universal Unique Identifier*), que se obtiene realizando una llamada al programa (suministrado como herramienta) *uuidgen*. La unicidad de este valor está asegurada dado que dicho identificador incluye el instante de creación y la ubicación (el computador) en la que se realizó. El compilador de IDL genera sustitutos (*stubs*) en C a partir de las interfaces, para ser ensamblados con las aplicaciones cliente y servidor.

La característica más potente de la RPC es que puede ser integrada con los servicios de seguridad y nombrado de DCE. Esta integración hace posible autenticar toda llamada a procedimiento y localizar los servidores dinámicamente en tiempo de ejecución.

4.2.3 Hilos

Los servidores pueden servir RPC concurrentes utilizando varios **hilos**. De la misma manera, un cliente puede necesitar varios hilos para atender una interfaz de usuario a la vez que realizan una RPC de un servidor.

4.2.4 Formato de los datos

La representación de datos en la red (NDR, *Network Data Representation*) de DCE especifica un estándar de formato de datos independiente de la arquitectura, con el fin de facilitar la transferencia de datos entre arquitecturas heterogéneas. Este esquema de codificación/decodificación aísla el código de las aplicaciones de las diferencias en los tipos de datos, facilitando la portabilidad e interoperabilidad de las aplicaciones.

4.2.5 DC++

DC++ [SM93] es un **entorno distribuido orientado a objetos** construido sobre DCE. En oposición a la visión procedimental de DCE (llamada a procedimiento remoto), soporta un modelo de objetos uniforme, invocación de objetos de grano fino independiente de la localización, paso de parámetros usando referencias remotas, migración dinámica de objetos e integración con el lenguaje C++. Los servicios fundamentales de DCE que utiliza DC++ son los hilos, las RPC y el CDS.

Los **objetos distribuidos** pueden ubicarse en diferentes nodos del sistema distribuido y manipulan referencias locales y remotas.

Las referencias remotas se implementan con una indirección proporcionada por un **representante** (*proxy*). Un representante contiene una pista de la localización del objeto referenciado y le reenvía las invocaciones de manera transparente utilizando RPC. Se instalará un representante en todo nodo que conozca la existencia de un objeto remoto. Este ocurre cuando se pasa una referencia a un objeto remoto como parámetro de una invocación. También ocurre cuando un objeto migra y tiene referencias a objetos remotos: en el nodo destino se tendrán que instalar representantes para todas las referencias.

Cada nodo mantiene una tabla de correspondencias entre **identificadores globales de objetos** (se utilizan los UUID de DCE) que vienen en las invocaciones entrantes y direcciones de almacenamiento reales de objetos C++.

Forman también parte del entorno uno o más servidores CDS, encargados de almacenar representantes de objetos que tienen registrado un nombre simbólico. De esta

manera, un objeto puede adquirir un representante de un objeto remoto proporcionando su nombre a un CDS.

La migración de objetos se solicita invocando un método generado automáticamente para todos ellos. Una vez movido el objeto, en el nodo original se dejará un representante del mismo. Como prerrequisito para la migración de un objeto, se asume que la clase a la que pertenece el objeto está disponible en el nodo destino basándose en un servicio de replicación de clases.

4.2.6 Crítica

Uso de paradigma procedimental

A pesar de que el uso de DCE está muy extendido, su enfoque procedimental excluye un funcionamiento orientado a objetos. Incluso aunque cliente y servidor estén construidos como conjuntos de objetos, su interacción pasa necesariamente por las llamadas a procedimientos.

De alguna manera, DC++ viene a solucionar dicho problema, proporcionando un entorno orientado a objetos. Además, aunque DC++ proporciona un entorno orientados a objetos, su construcción está basada en el carácter procedimental de DCE.

Descripción de servicios con IDL

Los servicios que proporcionan los servidores, en DCE, y los objetos distribuidos, en DC++, se tienen que describir utilizando el lenguaje de definición de interfaces IDL. El programador tiene, por tanto, que realizar una descripción de los servicios en un lenguaje (IDL) y la implantación de los mismos en otro (C++, en el caso de DC++).

Programación únicamente en C++

La construcción de aplicaciones, en el caso de DC++, está restringida al uso del lenguaje C++.

Capa de software

Cualquiera de las dos variantes (DCE y DC++) no deja de ser una capa de software o *middleware* que se ejecuta sobre un sistema operativo tradicional.

4.2.7 Características interesantes

Generación de identificadores únicos

Los identificadores globales y únicos de objetos permite establecer una correspondencia biunívoca entre ambos, de tal manera que los identificadores no pueden repetirse en el espacio (diferentes nodos) ni en el tiempo.

Pistas de localización en representantes en DC++

La utilización de pistas de localización almacenadas en los representantes de los objetos permite evitar búsquedas de objetos por difusión (*broadcast*) en muchos casos. Dado que se supone que los objetos se mueven con menos frecuencia que con la que son accedidos, en la mayor parte de los casos la pista de localización va a ser una indicación real de la ubicación del objeto.

Invocación independiente de la localización

Los objetos se comunican de manera uniforme a través de la invocación de métodos, con independencia de su ubicación. La tarea de localización de los objetos es realizada de manera transparente por DC++.

Migración de objetos de grano fino

Cualquier objeto, con independencia de su granularidad, puede migrar. La migración de objetos se realiza de manera transparente, de modo que el objeto puede reanudar su actividad en el momento en que es activado en el nodo destino.

4.3 CORBA

CORBA (*Common Object Request Broker Architecture*) [OMG99] es una especificación definida por el OMG (*Object Management Group*) para la creación y uso de objetos remotos, cuyo objetivo en proporcionar interoperabilidad entre aplicaciones en un entorno distribuido y heterogéneo.

Es conocido como un tipo de "*middleware*", ya que no realiza las funciones de bajo nivel necesarias para ser considerado un sistema operativo. A pesar de que debe funcionar sobre sistemas operativos tradicionales, realiza muchas de las operaciones que tradicionalmente se han considerado del dominio de los sistemas operativos para entornos distribuidos.

4.3.1 Objetos CORBA

Los **objetos CORBA** se diferencian de los objetos de los lenguajes habituales de programación en que:

- pueden estar localizados en cualquier lugar de la red,
- pueden ejecutarse en cualquier plataforma, y
- pueden estar escritos en cualquier lenguaje.

Un cliente puede utilizar un objeto CORBA sin saber donde está ni en qué lenguaje ha sido implementado.

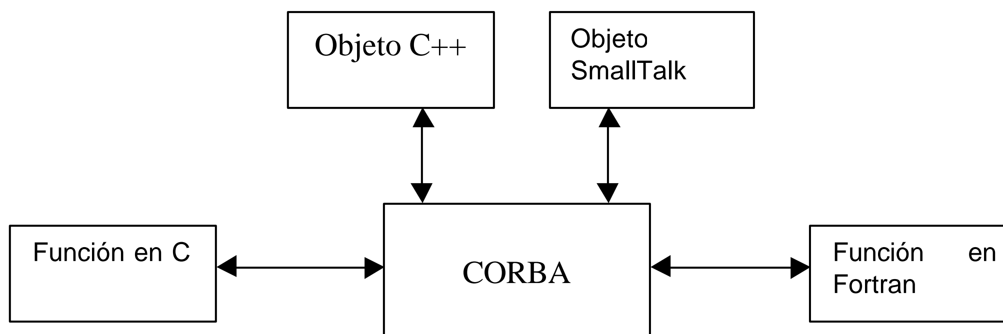


Figura 4.1. La función de CORBA.

4.3.2 El ORB

El corazón de CORBA es su ORB (*Object Request Broker*). El ORB es el responsable de:

- Buscar la implementación del objeto servidor.
- Prepararlo para recibir la petición.
- Poner en contacto el cliente con el servidor.

- Transportar los datos (parámetros y valores de retorno) entre uno y otro, transformándolos adecuadamente.

En definitiva es el responsable de que ni cliente ni servidor necesiten conocer ni la localización ni el lenguaje de implementación del otro.

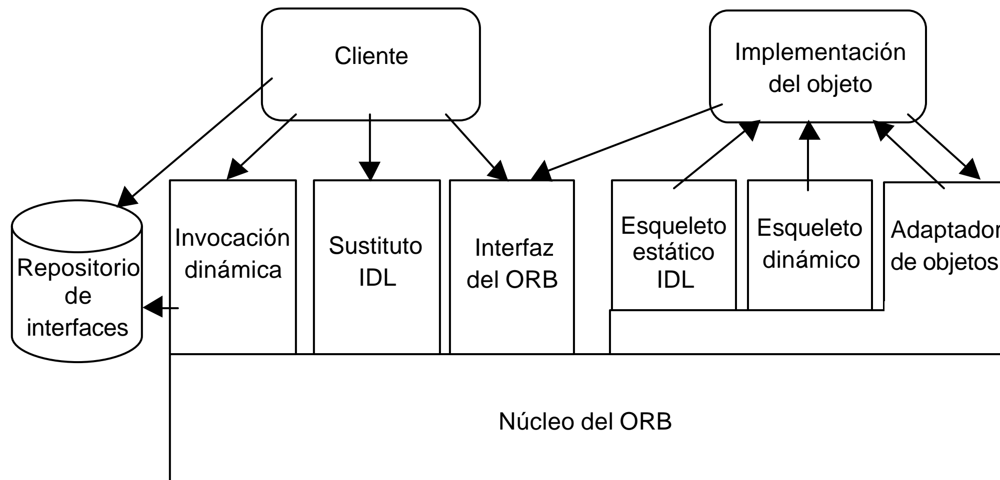


Figura 4.2. Estructura del ORB de CORBA.

4.3.3 Definición de objetos CORBA

Los objetos CORBA se tienen que definir con el **lenguaje de definición de interfaces IDL** (*Interface Definition Language*) que, como su propio nombre indica, se limita a definir la interfaz del objeto y no su implementación. La **definición del objeto** incluye la definición de sus atributos, sus métodos (denominados operaciones) y las excepciones que eleva. La **implementación del objeto** se tiene que realizar con algún lenguaje de programación que tenga enlaces con IDL (en la actualidad existen enlaces con lenguajes como C, C++, Java, Smalltalk, Ada, etc.).

IDL proporciona herencia múltiple de interfaces, de manera que las interfaces derivadas heredan las operaciones y los tipos definidos en las interfaces base. Todas las interfaces derivan de una interfaz raíz, denominada `Object`, la cual proporciona servicios que son comunes a todos los objetos CORBA, como duplicación y liberación de referencias a objetos, etc.

La compilación de una definición de objetos en IDL genera, entre otras cosas, un sustituto (*stub*) y un esqueleto (*skeleton*), para el cliente y servidor, respectivamente. El sustituto crea una petición de servicio al ORB a instancias del cliente y representa al objeto servidor. El esqueleto, por su parte, entrega las peticiones a la implementación del objeto CORBA.

Es posible utilizar para la definición de los objetos una serie de tipos basicos y construidos que no tienen la categoría de objetos y que son similares a otros tipos encontrados en la mayoría de los lenguajes de programación: *char*, *boolean*, *array*, *struct*, etc.

4.3.4 El repositorio de interfaces

Las interfaces de objetos se almacenan en un **repositorio de interfaces** (IR, *Interface Repository*), que proporciona un almacenamiento persistente de las declaraciones de interfaces realizadas en IDL. Los servicios proporcionados por un IR

permiten la navegación por la jerarquía de herencia de un objeto y proporcionan la descripción de todas las operaciones soportadas por un objeto.

La función principal del IR es proporcionar la información de tipos necesaria para realizar peticiones utilizando la **Interfaz de Invocación Dinámica**, aunque puede tener otros propósitos, como servir de almacenamiento de componentes reutilizables para los desarrolladores de aplicaciones.

4.3.5 La Interfaz de Invocación Dinámica

La compilación de las declaraciones IDL en algún lenguaje de programación permite a los clientes invocar operaciones en objetos conocidos, pero algunas aplicaciones necesitan poder realizar llamadas a objetos sin tener conocimiento de sus interfaces en tiempo de compilación. En esencia, la **Interfaz de Invocación Dinámica** (DII, *Dynamic Invocation Interface*) es un *stub* genérico de clientes capaz de enviar cualquier petición a cualquier objeto, interpretando en tiempo de ejecución los parámetros de la petición y los identificadores de la operación.

4.3.6 Adaptadores de objetos

CORBA permite que la implementación de los objetos sea todo lo variada que se quiera. En unos casos, varias interfaces IDL pueden estar implementadas por un solo programa, y, en otros casos, una interfaz IDL puede estar implementada por una serie de programas, uno para cada operación.

Un **Adaptador de Objetos** (OA, *Object Adapter*) proporciona los medios por los que varios tipos de implementaciones de objetos utilizan los servicios del ORB, como por ejemplo:

- generación de referencias de objetos.
- invocación de métodos de objetos.
- seguridad.
- activación y desactivación de objetos e implementaciones.

Dependiendo del ORB subyacente, un OA puede elegir entre proporcionar estos servicios delegando en el ORB o realizando el trabajo él mismo. En cualquier caso, las implementaciones de los objetos no van a estar al tanto de la diferencia, ya que únicamente usan la interfaz proporcionada por el OA.

4.3.7 Referencias

Para que un cliente pueda realizar una petición a un objeto servidor, deberá utilizar una **referencia** al objeto. Una referencia siempre refiere el mismo objeto para la que fue creada, durante tanto tiempo como exista el objeto. Las referencias son tanto inmutables como opacas, de manera que un cliente no puede “entrar” en una referencia y modificarla. Sólo el ORB sabe que es lo que hay “dentro” de la referencia.

Cuando se pasan objetos como parámetros en invocaciones a métodos, lo que realmente se pasan son referencias a dichos objetos. El paso de objetos como parámetro es, por tanto, por referencia.

4.3.8 Paso de parámetros por valor

En la última revisión importante de CORBA (revisión 2.3, de Junio de 1999) se introdujo la propuesta de **objetos-por-valor** (*objects-by-value*), que extienden el

modelo de objetos de CORBA tradicional para permitir el paso de objetos por valor dentro de los parámetros de los métodos. Para conseguirlo se introduce un nuevo tipo IDL denominado el tipo **valor** (*value*).

Cuando un ORB encuentra un objeto de tipo valor dentro de un parámetro, automáticamente pasa una copia del estado del objeto al receptor. En contraste, el ORB pasará por referencia cualquier objeto declarado vía una interfaz IDL.

Un tipo valor se puede entender a medio camino entre una interfaz IDL y una estructura (del estilo de las del lenguaje C). La sintaxis del valor permite especificar algunos detalles de implementación (por ejemplo, su estado), que no forman parte de una interfaz IDL. Además, se pueden especificar métodos locales para operar con dicho estado. A diferencia de las interfaces, los métodos de los tipos valor no pueden ser invocados remotamente.

4.3.9 Crítica

Falta de uniformidad

CORBA proporciona un modelo de objetos basado en las interfaces. Los clientes y servidores pueden programarse en diferentes lenguajes, sin que estén obligados a ser orientados a objetos. El programador debe trabajar, por tanto, de manera simultánea, con el modelo de objetos del lenguaje de programación (si es que es orientado a objetos) y el modelo de objetos proporcionado por CORBA.

El lenguaje IDL permite también la definición de entidades no objetos, que se corresponden con un conjunto predefinido de tipos básicos y tipos construidos.

Finalmente, con la introducción del paso de parámetros por valor, se introduce un nuevo tipo de objeto, el valor, con una semántica distinta a la de las interfaces, lo que perjudica aun más la uniformidad en la orientación a objetos.

Programación no transparente

A pesar de que los sustitutos y esqueletos ocultan de una manera importante los detalles de la interacción entre objetos, los clientes y servidores CORBA se tienen que construir de una manera concreta, para hacer posible su comunicación. En el código del usuario se mezclan partes relacionadas con el dominio del problema a resolver y otras relativas a la gestión de la comunicación.

Capa de software

A pesar de su gran éxito para la construcción de sistemas distribuidos, CORBA no deja de ser una capa añadida al sistema operativo.

Migración de objetos

El hecho de que los objetos puedan estar programados en diferentes lenguajes de programación y para diferentes arquitecturas y sistemas operativos imposibilita, prácticamente, su migración. Únicamente la utilización de CORBA con un lenguaje de programación independiente de la plataforma, como Java, permite cierto grado de movilidad a los objetos.

4.3.10 Características interesantes

Transparencia de localización y acceso

Los objetos remotos (los que están definidos por interfaces) son utilizados sin preocupación alguna por parte del programador sobre su ubicación. El programador

sólo debe conseguir una referencia a un objeto remoto para poder invocarlo, descansando en el ORB todas la responsabilidad de proporcionar dicha transparencia.

Uso de referencias globales

Una referencia a objeto es válida dentro del sistema distribuido con independencia de su propietario, de manera que si se pasa como parámetro en una invocación a método, la referencia seguirá siendo válida para el objeto invocado.

Excepciones en el modelo de objetos

Como ya se comentó en el capítulo anterior, las excepciones son un concepto muy importante para la construcción de programas robustos. Dada la naturaleza distribuida de los objetos CORBA, las excepciones podrán propagarse entre servidores y clientes con independencia de su ubicación.

4.4 DCOM

COM (*Component Object Model*) [Mic95] es la tecnología de definición y manipulación de componentes de Microsoft que proporciona un modelo de programación y un estándar binario para los mismos. DCOM [Mic98] (de *Distributed COM*) es la tecnología que extiende COM para permitir a los objetos componentes residir en máquinas remotas y está disponible desde la aparición de Windows NT 4.0. A partir de ahora se utilizarán los términos COM y DCOM indistintamente.

4.4.1 Modelo de objetos

Un **objeto COM** se define en términos de las interfaces individuales que soporta (una o más) y que están definidas en la clase (objeto clase) a la que pertenece. Cada interfaz está identificada por un **identificador único** (IID, *Interface Identifier*), que es un caso particular de **identificador global y único** (GUID, *Global Unique Identifier*). Los GUID son valores de 128 bits que se garantizan únicos estadísticamente en el espacio y en el tiempo.

Las interfaces son el único medio de interactuar con un objeto COM. Un cliente que desea utilizar los servicios de un objeto habrá de obtener primero un puntero a una de sus interfaces.

Los **objetos clase** implementan una o más interfaces y se identifican por un **identificador de clase** (CLSID, *Class Identifier*). Los CLSID son también un caso particular de GUID. Con el fin de que un cliente pueda crear un objeto COM, es necesario describir su clase utilizando el **lenguaje de definición de interfaces** (IDL, *Interface Definition Language*). La compilación de dicha descripción genera un representante (*proxy*) para los clientes y un sustituto (*stub*) para el servidor.

COM no proporciona la herencia como instrumento para lograr la reutilización. En su lugar proporciona los **mecanismos de contención y agregación**. El polimorfismo es conseguido cuando diferentes clases soportan la misma interfaz, permitiendo a una aplicación utilizar el mismo código para comunicarse con cualquiera de ellas.

4.4.2 Interoperabilidad entre objetos COM

El objetivo principal de COM es proporcionar un medio por el que los clientes pueden hacer uso de los objetos servidores, sin tener en cuenta que pueden haber sido desarrollados por diferentes compañías utilizando diferentes lenguajes de programación. Con el fin de lograr este nivel de interoperabilidad, COM define un **estándar binario**,

que especifica cómo se dispone un objeto en memoria principal en tiempo de ejecución. Cualquier lenguaje que pueda reproducir dicha disposición en memoria podrá crear objetos COM.

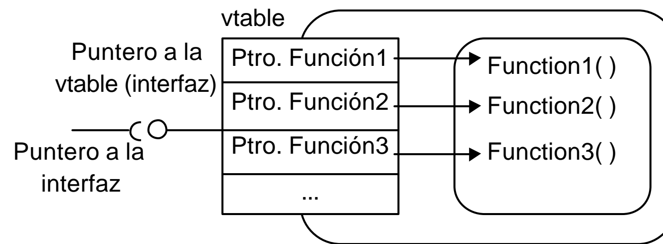


Figura 4.3. Disposición de un objeto COM en memoria.

Además del objetivo de la interoperabilidad, COM tiene otros objetivos:

- Proporcionar una solución para los problemas de versiones y evolución.
- Proporcionar una visión del sistema de los objetos.
- Proporcionar un modelo de programación singular.
- Proporcionar soporte para capacidades de distribución.

4.4.3 Modelo de programación

En el modelo de programación COM, los clientes COM se conectan a uno o más objetos COM. Cada objeto COM expone sus servicios a través de una o más interfaces, que no son más que agrupaciones de funciones relacionadas semánticamente.

La implementación compilada de cada objeto COM está contenida en un módulo binario (`exe` o `dll`) denominado **servidor COM**. Un único servidor COM es capaz de contener la implementación compilada de varios objetos COM.

Un servidor COM puede estar enlazado al proceso cliente (*in-process server*), puede ejecutarse en un proceso distinto del cliente pero en la misma máquina (*local server*) o bien, puede ejecutarse en un proceso separado en una máquina distinta, incluso en un sistema operativo distinto (*remote server*). Para la comunicación con objetos situados en espacios de direcciones distintos del espacio de direcciones del cliente, se utilizan intermediarios en la forma de representantes y sustitutos.

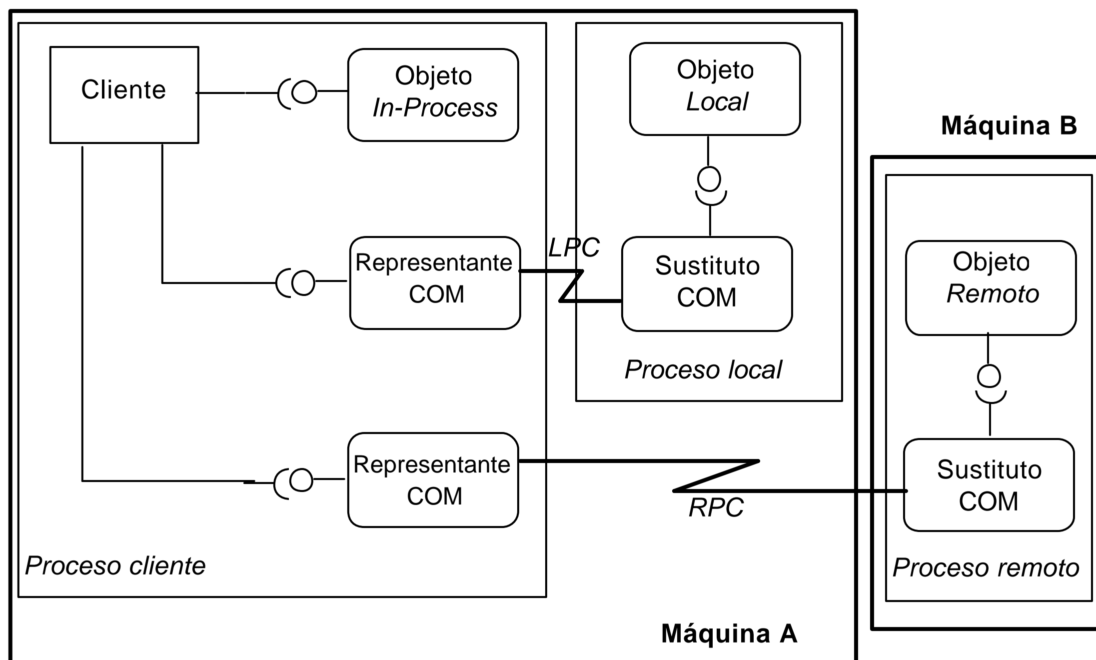


Figura 4.4. Diferentes disposiciones de un objeto COM respecto a un cliente.

El modelo de programación COM define que un servidor COM debe exponer objetos COM, un objeto COM debe exponer sus servicios y un cliente COM debe usar los servicios de objetos COM.

Para comunicarse con un objeto que no es local, COM emplea un mecanismo de comunicación entre procesos que es transparente a la aplicación, incluso en lo relativo a la localización del objeto.

4.4.4 Ciclo de vida

Todos los objetos COM tienen que implementar una interfaz particular, denominada **IUnknown**. Esta interfaz es el corazón de COM y es utilizada para negociación de interfaces en tiempo de ejecución (preguntar al objeto qué interfaces soporta y obtener punteros a las mismas), gestión del ciclo de vida del objeto y agregación.

Cada objeto mantiene una **cuenta de referencia** que es incrementada cada vez que un cliente solicita un puntero para una interfaz o pasa una referencia al mismo. Cuando la interfaz deja de ser utilizada por el cliente, la cuenta de referencia se decrementa. Las operaciones de incremento y decremento son realizadas de manera explícita por el cliente invocando funciones pertenecientes a la interfaz *IUnknown*.

4.4.5 COM+

COM+ [Kir97] es la nueva generación de COM y se presenta como la base del nuevo sistema operativo de Microsoft, Windows 2000.

COM+ viene a ocultar la mayor parte de las tareas que en COM resultaban tediosas y dificultosas para los programadores, como el control del ciclo de vida, la negociación de interfaces, etc. A pesar de todo, el modelo básico de objetos sigue siendo el de COM.

El conjunto de servicios que introduce COM+ está orientado a la construcción de aplicaciones empresariales, de tal forma que los programadores se concentran en la

escritura de la lógica de negocio y no tienen que perder tiempo escribiendo infraestructura u otros servicios.

Las características principales que se pueden encontrar en COM+ son: servidores, transacciones, seguridad, administración, equilibrado de carga, componentes encolados (*queued components*) y eventos.

4.4.6 Crítica

Estándar binario

COM proporciona un concepto de objeto muy vago y alejado del utilizado habitualmente y descrito en el capítulo anterior. En primer lugar, la herencia no está contemplada en el modelo de objetos para la reusabilidad, aunque se proporcionan otros mecanismos. En segundo lugar, los objetos no tienen una identidad propiamente dicha, sino que solamente existen para los clientes en tanto en cuanto se disponga de algún puntero a alguno de sus interfaces.

Los objetos necesitan procesos

Los objetos no son entidades autónomas, sino que necesitan de los procesos, que les proporcionan un entorno en el que ejecutarse.

Dificultad en la construcción de programas

A pesar de que COM+ viene a solucionar, en parte, las dificultades inherentes a la programación con objetos COM, la programación sigue siendo compleja. Los programadores de objetos COM (servidores) deben realizar tareas adicionales a las propias del dominio del problema, con el fin de que su objeto pueda ser utilizado por los clientes. Los clientes, por su parte, sólo pueden utilizar punteros para referenciar los objetos a través de alguna de sus interfaces.

4.4.7 Características interesantes

Transparencia de localización y acceso

COM se encarga de proporcionar a los clientes y objetos servidores la transparencia necesaria para las invocaciones locales y remotas. Una vez que un cliente obtiene un puntero a una interfaz de un objeto, puede invocar sus funciones con independencia de su ubicación. De la misma forma, no es necesario que el programador realice ninguna tarea para el tratamiento de las invocaciones locales y/o remotas.

Integración del modelo de objetos en el sistema operativo

COM es utilizado no solamente para proporcionar un modelo de objetos distribuidos al programador, sino también para que el sistema operativo proporcione sus servicios a los programadores en forma de objetos. Evidentemente, dicha integración se está llevando a cabo únicamente en los sistemas operativos de Microsoft.

4.5 RMI de Java

RMI (*Remote Method Invocation*) [Sun98] fue diseñado para permitir la invocación de métodos remotos de objetos entre distintas máquinas Java de manera transparente. Integra directamente un modelo de objetos distribuidos en el lenguaje Java a través de un conjunto de clases e interfaces.

4.5.1 Objetos RMI

Un **objeto RMI** es un objeto cuyos métodos pueden ser invocados desde otra máquina Java, incluso a través de una red. Cada objeto remoto implementa una o más **interfaces remotas** que especifican qué operaciones pueden ser invocadas por los clientes. Cualquier otra operación pública que tenga el objeto pero que no aparezca en la interfaz no podrá ser utilizada por los clientes remotos. Los clientes invocan dichos métodos exactamente igual que si fueran métodos locales, quedando ocultos los detalles de la comunicación. Se utilizan los denominados **sustitutos** (*stub*) y **esqueletos** (*skeleton*), que actúan de intermediarios entre los objetos local y remoto. Los sustitutos y los esqueletos son generados de manera automática por el compilador *rmic*.

4.5.2 Modelo de programación

Desde el punto de vista del programador, los objetos remotos se manipulan de la misma manera que los locales, a través de referencias. Realmente, una **referencia** a un objeto remoto apunta a una referencia a un sustituto local que lo representa, y que es el encargado de transmitir los parámetros de la invocación al computador remoto y recibir de él el valor de retorno. La manipulación de objetos remotos tiene que tener en cuenta:

- Cómo obtener una referencia a un objeto remoto.
- Es necesario manejar excepciones específicas de la invocación de métodos remotos.

Por su parte, un esqueleto es el encargado de recoger los parámetros que recibe del sustituto a través de la red, invocar de manera local al objeto remoto y devolver el resultado de nuevo a través de la red al sustituto del objeto que realizó la invocación.

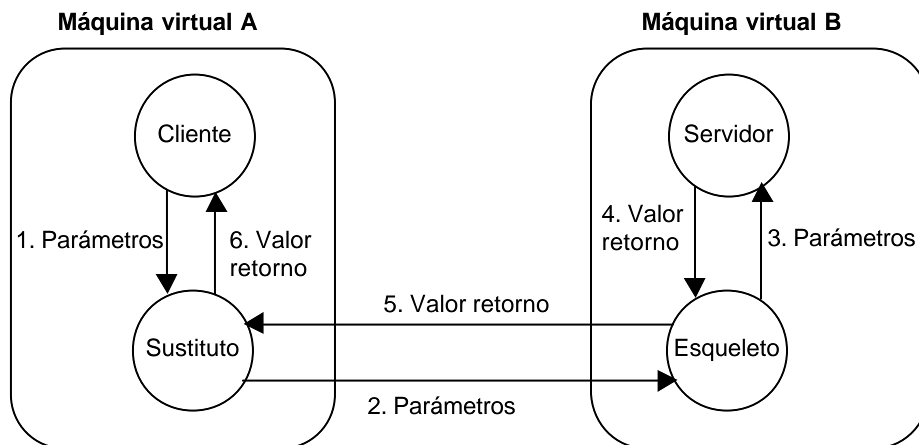


Figura 4.5. Uso de sustitutos y esqueletos en la invocación remota.

A diferencia de una invocación local, una invocación RMI pasa los objetos locales que forman parte de la lista de parámetros por copia (por valor), dado que una referencia a un objeto local sólo sería útil en una máquina virtual única. RMI utiliza el **servicio de serialización de objetos** [**REFERENCIA**] para aplanar el estado de un objeto local y colocarlo en el mensaje a enviar a la máquina virtual remota. Si el objeto es no-serializable, no se puede usar como parámetro. También son pasados por valor los parámetros de tipos primitivos. Por otro lado, RMI pasa los objetos remotos (objetos que implementan una interfaz remota) por referencia.

4.5.3 Servicios

RMI proporciona interfaces y clases para buscar objetos remotos, cargarlos y ejecutarlos de manera segura. Adicionalmente, proporciona un **servicio de nombrado** un tanto primitivo (servicio de nombrado no persistente y espacio de nombres plano) que permite localizar objetos remotos y obtener referencias a ellos, de manera que se puedan invocar sus métodos.

La utilización de objetos remotos lleva consigo la aparición de nuevas situaciones de error, de tal forma que el programador deberá manejar el conjunto de nuevas excepciones que pueden ocurrir durante una invocación remota.

RMI incluye una característica de recolección de basura distribuida, que recolecta aquellos objetos servidores que no son referenciados por ningún cliente de la red.

4.5.4 Crítica

Interoperabilidad únicamente entre objetos Java

RMI sólo existe en el lenguaje Java. Por tanto, la interoperabilidad que ofrece entre objetos distribuidos se circunscribe únicamente a objetos Java, al contrario que CORBA o DCOM, por ejemplo.

Semántica diversa en el paso de parámetros

En el paso de parámetros se mezclan dos semánticas, paso por valor y paso por referencia, que dependen del tipo de la entidad: objeto local, objeto remoto o tipo primitivo. En el caso de que el programador desee una semántica concreta para un parámetro, deberá escribir el código necesario para que el parámetro con la información deseada tenga el tipo oportuno.

Falta de transparencia

Los objetos remotos se programan de manera diferente a los objetos locales, dado que tienen que implementar una interfaz determinada y heredar de una clase concreta que le permiten obtener la funcionalidad necesaria para ser invocados remotamente.

4.5.5 Características interesantes

Transparencia de localización y acceso

Una vez que un objeto cliente obtiene una referencia a un objeto remoto, lo utiliza como si fuera local, invocando sus métodos de la forma habitual. La única precaución que tiene que tomar el programador es la forma en que dispone los parámetros de la invocación debido a la variedad semántica del paso de parámetros.

Excepciones distribuidas

El modelo de excepciones incluido en Java queda automáticamente extendido para el caso de los objetos distribuidos, de manera que las excepciones generadas en un objeto remoto se propagan al objeto cliente incluso si residen en computadores diferentes.

4.6 Guide

Guide [BBD+91], nacido en el instituto INRIA, en Francia, constituye un sistema que pretende dar soporte a aplicaciones distribuidas cooperativas. Se intenta ofrecer un universo distribuido compartido, organizado como un conjunto de objetos compartidos

por actividades concurrentes. Las diferentes actividades o procesos se comunican entre sí a través del uso de operaciones de objetos situados en otras actividades. Se trata, por tanto de un sistema operativo que pretende dar solución al problema de la interoperabilidad entre objetos, surgido recientemente con la expansión de la tecnología de la orientación a objetos y la distribución.

El sistema constituye una plataforma para lenguajes de programación orientados a objetos como el C++ y el lenguaje Guide. Se pretende dar soporte para que aplicaciones orientadas a objetos implementadas en estos lenguajes, puedan establecer comunicación entre sí de manera segura, sin confiar en el código de cada uno.

Guide puede verse con una máquina virtual distribuida y multiprocesador. La distribución no es visible a los usuarios aunque sí lo es el paralelismo. Las aplicaciones se estructuran en objetos, que se almacenan de manera transparente en almacenamiento secundario y serán cargados bajo demanda en almacenamiento volátil para su ejecución.

4.6.1 Modelo de objetos

Guide soporta un modelo de **objetos pasivo**, de tal forma que un objeto encapsula únicamente datos y operaciones. Todos los objetos son persistentes y tienen un nombre único, global e independiente de su localización denominado **referencia del sistema**. Las referencias del sistema no son visibles a los usuarios y son utilizadas internamente para la invocación de objetos.

La distribución de los objetos no es visible para el programador, que no necesitará saber dónde está localizado un objeto particular. Así mismo, los objetos pueden migrar entre diferentes nodos.

4.6.2 Modelo de ejecución

La **tarea** (*job*) es la unidad de ejecución. Una tarea es un espacio virtual potencialmente distribuido y está compuesta de un conjunto de hilos de control concurrentes denominados **actividades**, que operan en los objetos pasivos. Para arrancar una aplicación, el usuario tiene que especificar un objeto inicial y un método inicial. Se crea entonces una tarea, asociando el objeto inicial a la misma y creándose una actividad para la invocación al método inicial.

Una tarea puede extenderse sobre varios nodos, creciendo o decreciendo el número de estos de manera dinámica, de acuerdo al patrón de invocaciones a objetos. La ejecución de una actividad consiste en invocaciones sucesivas a métodos de objetos, pudiendo tener lugar dicha invocación en cualquier nodo del sistema. Durante cada invocación, el objeto referenciado es localizado y asociado dinámicamente a la memoria virtual de la tarea. El conjunto de todos los objetos asociados a una tarea se denomina **contexto** de la tarea.

No existe una comunicación explícita entre actividades. La única forma que tienen de conseguirlo es a través de invocaciones a objetos compartidos.

4.6.3 Invocación de objetos

La operación de **invocación de objetos** debe especificar la referencia del sistema del objeto invocado (su nombre interno), el nombre del método y los parámetros de la invocación. La referencia contiene un **identificador único** de objeto (*oid*) y alguna **pista de su ubicación**.

Una **invocación remota** es aquella que tiene lugar en un nodo distinto de aquel en el que se realizó la invocación. En ese caso, la tarea y la actividad se difunden al nodo remoto

y el objeto es invocado allí. En el caso de que el objeto invocado se encuentre únicamente en el almacenamiento persistente, deberá ser cargado en el almacenamiento volátil, eligiéndose el correspondiente al nodo desde el que se realizó la invocación (Guide no incluye equilibrado de carga). El objeto será cargado en el nodo en el que fue creado únicamente si fue especificado como inmóvil (*unmovable*).

4.6.4 Migración de objetos

Los mecanismos de migración que se proporcionan en Guide están diseñados para ser utilizados en tiempo de administración, es decir, cuando la aplicación no está ejecutándose. No es posible, por tanto, que los objetos migren dinámicamente, mientras están siendo utilizados.

4.6.5 Crítica

Objetos pasivos

El sistema operativo Guide tiene un modelo de objetos pasivo, de tal forma que se diferencian las entidades activas, los procesos, de las entidades pasivas, los objetos, que son meros contenedores de datos y métodos. Como ya se comentó en el capítulo anterior, el modelo de objetos pasivo rompe la uniformidad del sistema, al tener que gestionarse dos tipos de entidades claramente diferentes, y dificulta la migración de objetos notablemente.

Lenguajes de programación

Los únicos lenguajes de programación disponibles son el C++ y el Guide. La interoperabilidad entre objetos se restringe, por tanto, a objetos creados con estos lenguajes.

Migración estática de objetos

El mecanismo de migración está muy limitado, dado que no permite que los objetos se muevan dinámicamente a la vez que están siendo utilizados por las aplicaciones. Una de las consecuencias principales de la falta de migración dinámica es la ausencia de equilibrado de carga en el sistema.

Ausencia de equilibrado de carga

Un sistema operativo distribuido que permite la migración de objetos debe proporcionar algún mecanismo para el equilibrado de carga. Su ausencia es un handicap para conseguir un buen rendimiento en situaciones en las que hay un gran desequilibrio en la carga de computación de los diferentes nodos.

4.6.6 Características interesantes

Pistas en las referencias

La utilización en las referencias de pistas sobre la ubicación del objeto referenciado es muy útil en la invocación remota, dado que en muchos casos va a ser una información válida. Dado que se espera que los objetos migren con menos frecuencia que con la que son invocados, las pistas de ubicación podrán ser utilizadas en lugar de operaciones de localización más costosas como la difusión.

Distribución no visible al programador

El programador puede concentrarse en el código correspondiente al dominio del problema, dado que va a ser totalmente ajeno a la ubicación de los objetos involucrados en el mismo.

Activación transparente de los objetos persistidos

Aquellos objetos que son invocados y se encuentran en el almacenamiento de persistencia, se activan automáticamente. En el caso de que la activación se realizase en un nodo distinto al nodo desde el que se realizó la invocación, la invocación remota resultante se realizaría también de manera transparente.

4.7 Clouds

Clouds es un sistema operativo pensado para dar soporte a objetos distribuidos [DAM+90, DLA+91] desarrollado en el Instituto de Tecnología de Georgia. Está implementado sobre el micronúcleo Ra, que funciona sobre máquinas Sun. Puede ser utilizado para aplicaciones centralizadas y distribuidas que pueden ser escritas con lenguajes orientados y no orientados a objetos.

4.7.1 Abstracciones de Clouds

Clouds utiliza la abstracción de **hilo** para la computación y la de **objeto** para representar el espacio de almacenamiento.

Un objeto Clouds es un **objeto pasivo de grano grueso**, que equivale a un espacio de direcciones virtuales de un proceso convencional. En este espacio se almacenan los datos y el código del objeto.

La estructura interna de un objeto no es conocida por el sistema. Puede ser internamente una colección de objetos programados en C++, pero estos objetos internos no pueden ser utilizados desde fuera del objeto Clouds. A pesar de que el usuario puede crear estos objetos Clouds, **no existe el concepto de clases ni de herencia** en este sistema.

Los objetos son globales y como tales tienen un **nombre único** (*sysname*) siempre válido dentro del sistema distribuido. La utilización de este identificador único permite la invocación transparente de un objeto independientemente de su localización. Los usuarios pueden definir nombres de más alto nivel para los objetos, que serán traducidos a los nombres de bajo nivel con un **servidor de nombres**.

Los **hilos** no están asociados a ningún objeto concreto, y van ejecutándose en los espacios de direcciones de los objetos, viajando de uno a otro a medida que se van invocando métodos de los distintos objetos. Los hilos pueden tener unas etiquetas especiales que permiten mantener diferentes tipos de atomicidad y consistencia en la invocación a operaciones de los objetos.

4.7.2 Distribución en Clouds

El sistema operativo Clouds se ejecuta sobre un conjunto de nodos que juegan uno o varios de los siguientes papeles: servidor de cómputo, servidor de datos o estación de usuarios.

Los **servidores de cómputo** proporcionan la potencia de computación. Los **servidores de datos** funcionan como un repositorio para los datos (objetos) persistentes. Las **estaciones de usuarios** proporcionan un entorno de programación a

los usuarios y sirven de interfaz con los servidores de cómputo y de datos. Todos los objetos se almacenan en servidores de datos, pero son accesibles desde todos los servidores de cómputo del sistema de una manera transparente.

El **núcleo** del sistema está formado por un conjunto de máquinas homogéneas del tipo servidor de cómputo, que no tienen almacenamiento secundario y que proporcionan el servicio de ejecución para los hilos.

4.7.3 Espacio de objetos

Los objetos Clouds forman un **espacio de objetos disponible y accesible desde cualquier máquina** del sistema, a la manera de una memoria global compartida. La compartición de esta memoria global se proporciona con un mecanismo denominado **Memoria Compartida Distribuida** (DSM, *Distributed Shared Memory*).

Si un hilo en un nodo X invoca un método de un objeto O, la ejecución del método invocado tendrá lugar en el nodo X. Si O no está ubicado en el nodo X, se producirá una serie de faltas de página que serán servidas leyendo páginas de O desde el servidor de datos en el que reside actualmente. Únicamente las partes necesarias del código y los datos de O serán llevados a X. Este esquema permite que todos los objetos sean accesibles desde todos los servidores de cómputo. Se utiliza adicionalmente un protocolo de coherencia de la DSM para que varios hilos que se estén ejecutando en el mismo objeto utilicen de manera consistente los datos del mismo, incluso si se están ejecutando en diferentes servidores de cómputo.

La conclusión del uso del mecanismo de memoria compartida distribuida es que todo objeto del sistema reside lógicamente en cada uno de sus nodos. Este concepto permite separar el almacenamiento de los objetos de su uso.

4.7.4 Crítica

Modelo de objetos restringido

El modelo de objetos que soporta el sistema está muy alejado del modelo mencionado en el capítulo anterior. No soporta el concepto de clases, ni el de herencia. Los objetos son de un grano muy grueso, muy alejado del grano fino de los objetos de las aplicaciones. La estructura interna de objetos dentro de un objeto Clouds es totalmente desconocida para el sistema. El objeto Clouds es una pequeña evolución de un espacio de direcciones (objeto) sobre los que pueden funcionar los proceso (hilos que pasan por el objeto).

4.7.5 Características interesantes

Replicación de objetos soportada por el sistema operativo

El propio sistema operativo ofrece, de manera transparente, la replicación de aquellos objetos que están siendo invocados por parte de diferentes hilos, e incluso en diferentes nodos. Se asegura, además, la coherencia de los datos replicados.

Identificador global de objetos para transparencia de localización

Otro aspecto interesante es el uso de un identificador global de objetos, que permite la invocación transparente de un objeto independientemente de su localización.

4.8 Spring

Spring [MGH+94] es un sistema operativo distribuido, orientado a objetos y muy modular desarrollado por Sun, centrado en el desarrollo de interfaces fuertes entre los distintos componentes del sistema operativo, y que trata dichos componentes como partes reemplazables.

Los recursos del sistema se representan como objetos y todas sus interfaces se definen con un **lenguaje de definición de interfaces** (IDL, *Interface Definition Language*). Las interfaces únicamente especifican qué hacen los objetos, pero no cómo están implementadas sus operaciones.

Adicionalmente, el sistema operativo está estructurado sobre un micronúcleo (denominado **nucleus**), que implementa la mayor parte de la funcionalidad del sistema (sistemas de ficheros, paginadores, software de red, etc.) como servicios de nivel aplicación que se ejecutan sobre dicho micronúcleo, excepto el gestor de memoria virtual, que se ejecuta en modo privilegiado (*kernel*).

Spring no especifica cómo se deben implementar los objetos. Cada aplicación puede implementar los objetos de una manera diferente, llegando incluso a poder optar por diferentes maneras de invocar objetos.

4.8.1 Abstracciones

El micronúcleo soporta tres abstracciones básicas:

- **Dominios** (*domain*): análogos a los procesos de Unix, proporcionan un espacio de direcciones para la ejecución de aplicaciones y funcionan como contenedores de recursos, como los hilos (*thread*) y las puertas (*door*).
- **Hilos** (*thread*): se ejecutan dentro de dominios. En general, todo dominio es multihilo. Cada hilo individual se encarga de realizar una parte del trabajo global de la aplicación.
- **Puertas** (*door*): soportan las llamadas orientadas a objetos entre diferentes dominios. Una puerta describe un punto de entrada particular a un dominio. De alguna manera, recuerda los *sockets* de Unix BSD [LMK+89] o los puertos de Mach [ABB+86].

4.8.2 Invocación de objetos

Spring solamente proporciona mecanismos para la invocación de objetos en diferentes dominios. La forma en que se resuelve la invocación de objetos dentro de un mismo dominio es responsabilidad de lenguaje de programación con el que se ha implementado la aplicación.

Cuando un hilo en un dominio desea realizar una invocación de un objeto en otro dominio, tendrá que hacerlo a través de una puerta del dominio invocado, que permite identificar el objeto a invocar. En este caso, el micronúcleo crea un hilo en el dominio invocado, le transfiere el control y le pasa toda la información que necesita para atender la invocación. Cuando finaliza la ejecución del hilo del objeto invocado, el micronúcleo desactiva dicho hilo y reactiva el hilo que realizó la invocación, pasándole los datos devueltos por el hilo invocado.

Para proporcionar invocación de objetos localizados en diferentes nodos del sistema, el mecanismo de invocación se extiende con el uso de **representantes de red** (*network proxy*), que conectan los micronúcleos de diferentes máquinas de una manera

transparente. Estos representantes no son más que dominios normales, no recibiendo soporte especial del micronúcleo y hablarán un protocolo de red determinado. Con el fin de poder trabajar con diferentes protocolos, una máquina Spring podrá disponer de varios representantes.

Los representantes se encargan de reenviar invocaciones a puertas entre dominios de diferentes máquinas, haciendo transparente a los dominios cliente y servidor su existencia.

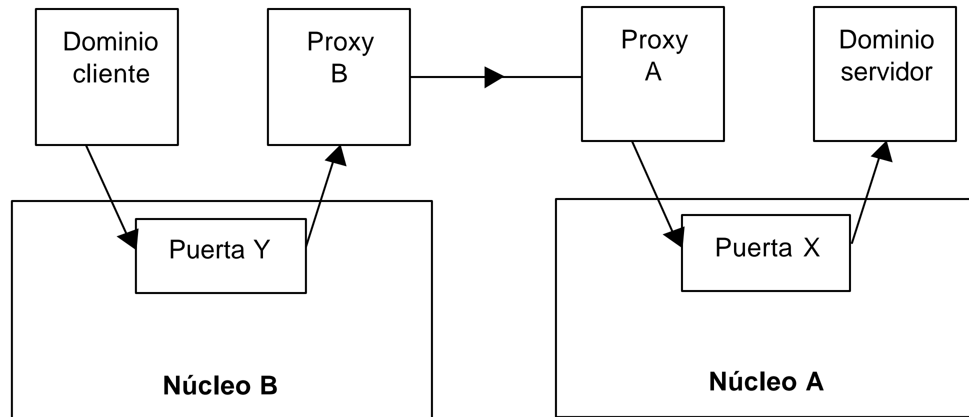


Figura 4.6. Uso de representantes para posibilitar las invocaciones remotas.

4.8.3 Nombrado

Spring proporciona un servicio de nombres uniforme. Todo objeto puede tener un nombre. Los nombres pueden pertenecer a contextos y los contextos a su vez pueden contener otros contextos, de tal forma que se puede construir un espacio de nombres jerárquico. Los contextos y los nombres son a su vez objetos Spring.

4.8.4 Seguridad

El acceso a los recursos está regulado por un mecanismo de seguridad. En Spring el acceso seguro a objetos se proporciona con un modelo de capacidades software y un modelo de listas de control de acceso.

4.8.5 Crítica

Modelo de objetos elemental

El modelo de objetos de Spring no soporta mecanismos de reutilización como herencia, polimorfismo, etc., mínimamente exigibles a un sistema de gestión de objetos. Los objetos Spring parecen, más bien, tipos abstractos de datos (TAD).

Falta de uniformidad en la orientación a objetos

Los objetos Spring residen en espacios de direcciones (dominio) en los que está implementada su funcionalidad. Spring es ajeno a la forma en que los objetos se disponen en los dominios y la manera en que se proporciona dicha funcionalidad, por lo que coexistirán (posiblemente) varios modelos de objetos: el de Spring, basado en dominios, hilos y puertas, y el de la implementación de los objetos (si se utiliza algún lenguaje orientado a objetos).

La falta de uniformidad afecta también a la forma en que se invocan los objetos. La invocación de objetos en diferentes dominios se realiza utilizando los mecanismos que

proporciona Spring. La invocación dentro del mismo dominio no está especificada, y queda determinada por el lenguaje de programación utilizado. El programador tiene que estar al tanto de estas dos maneras de utilizar los objetos en función de su ubicación.

Falta de soporte para la migración

Spring no proporciona ningún tipo de soporte para la migración de objetos o dominios, lo que impide tareas como el equilibrado dinámico de carga entre los diferentes nodos del sistema.

4.8.6 Características interesantes

Uso de representantes de red

Los representantes de red posibilitan la invocación transparente de objetos que residen en dominios remotos. Su independencia del núcleo es muy importante, dado que permite su sustitución o modificación sin afectar al sistema operativo.

Relación entre hilos y dominios

Los hilos se crean dentro de los dominios cuando un objeto que reside en uno de ellos recibe una invocación externa. Este tipo de relación entre objetos e hilos recuerda el modelo de objetos activo, mencionado en el capítulo anterior y sobre el que ya se han comentado una serie de ventajas para la distribución.

4.9 SOS

El sistema operativo SOS [SGH+89] es un sistema operativo distribuido que da soporte de objetos, desarrollado por el INRIA dentro del proyecto Esprit SOMIW (*Secure Open Multimedia Integrated Workstation*).

4.9.1 Modelo de objetos

El modelo de objetos soportado por el sistema operativo es a la vez sencillo y potente. Un **objeto elemental** es un conjunto de datos y código definido por el usuario. Los objetos elementales no son conocidos por el sistema, siempre y cuando no se desee migrarlos, almacenarlos o que sean accesibles de manera remota. Los objetos SOS sí son conocidos por el sistema y a ellos se referirá el resto de la exposición.

Todos los objetos pertenecen a un **contexto**, que es un espacio de direcciones. Un objeto SOS puede migrar entre contextos, y, en un momento dado, o bien está activo en un contexto, o está almacenado en disco.

Todo objeto tiene un **identificador único**, denominado **OID**. Desde dentro del contexto, un objeto es referenciado por su dirección, y, globalmente, por una referencia, que contiene un OID y una pista de su ubicación.

4.9.2 Distribución de objetos

SOS está diseñado para favorecer el uso del concepto de **representante** (*proxy*) [Sha86] para la estructuración de aplicaciones distribuidas. Los representantes permiten extender el concepto de objeto al de objeto distribuido u **objeto fragmentado**. Externamente, un objeto fragmentado parece un objeto simple. Sus fragmentos locales o representantes, que son objetos elementales, proporcionan su interfaz. Internamente, todos los posibles fragmentos estarán distribuidos.

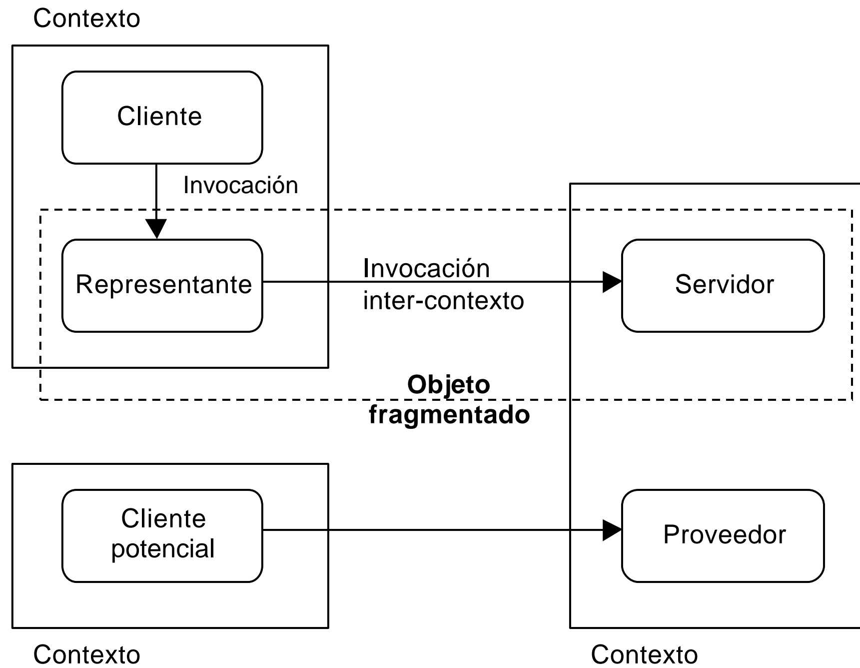


Figura 4.7. Concepto de representante: un servicio implementado como un objeto fragmentado.

Un servicio distribuido se implementa como un objeto fragmentado. Un cliente puede acceder al servicio invocando un fragmento local. Para que un cliente pueda conseguir acceso a un nuevo servicio, deberá solicitar al **gestor de objetos fragmentados** (vía su representante local) un representante para dicho servicio.

4.9.3 Migración de objetos

Un mecanismo básico de SOS es la migración de objetos. Un objeto que migra lleva consigo una lista de **prerrequisitos**, en la cual consta el conjunto de objetos que deben estar presentes siempre en su entorno. En caso de que sea necesario, el sistema migrará a su vez los representantes necesarios de dicho conjunto.

SOS utiliza una **interfaz de retrollamadas** (*upcall*) que permite a los objetos imponer su propia política sobre el mecanismo de migración del sistema. El método del objeto que se invoca con la retrollamada tomará decisiones como por ejemplo si la migración se realizará por movimiento o por copia, si le afectará sólo a él o a algún otro objeto, etc. Finalizada la migración, una retrollamada permitirá al objeto restablecer su entorno.

4.9.4 Modelo genérico

Todos los mecanismos mencionados son genéricos e independientes del lenguaje de programación. Para que SOS soporte objetos de un lenguaje de programación cualquiera, basta con asociar los prerrequisitos apropiados y los métodos invocados por las retrollamadas.

4.9.5 Diseño del sistema operativo

SOS está construido utilizando sus propios mecanismos: todos los servicios del sistema SOS están implementados como objetos fragmentados con representantes

locales. Entre ellos destacan el gestor de objetos, un servicio de nombrado flexible y un servicio de comunicaciones orientado a objetos.

4.9.6 Crítica

Coexistencia de varios modelos de objetos

Como ocurre en otros sistemas operativos ya revisados, en SOS coexisten dos modelos de objetos: el del propio sistema SOS u objeto fragmentado, que es conocido por el resto del sistema, y el de los objetos elementales, opacos a SOS y al resto de los objetos del sistema que no residan en su mismo contexto.

Falta de transparencia de localización

Los objetos se referencian de manera distinta según su ubicación respecto al objeto que los utiliza. Para referirse a un objeto que está en el mismo contexto, se utiliza su dirección; si el objeto está en otro contexto, se utiliza una referencia, que contiene su identificador de objeto y una pista de su ubicación.

4.9.7 Características interesantes

Uniformidad en la construcción del sistema

Tanto el propio sistema operativo como las aplicaciones de los usuarios se basan en la utilización de los objetos fragmentados con representantes. Dicha uniformidad permite a los objetos de usuario interactuar entre ellos y con los del sistema operativo de una manera común.

Utilización de pistas de localización de los objetos

Como ocurre en el sistema operativo Guide, la utilización de pistas en las referencias permite optimizar las operaciones de búsqueda de los objetos, ya que, en muchas ocasiones, las pistas proporcionarán información totalmente válida sobre la ubicación del objeto.

Migración conjunta de objetos

La lista de prerrequisitos para la migración permite identificar qué grupos de objetos tienen que moverse de manera solidaria, dado que se les supone una fuerte interrelación. La migración solidaria permite maximizar el número de invocaciones a objetos que se resuelven de manera local.

Especificación por el usuario de la política de migración

La separación de políticas y mecanismos de un sistema distribuido es muy importante para facilitar al máximo su adaptabilidad. La interfaz de retrollamadas permite modificar la política de migración que por defecto proporciona SOS. Esta posibilidad es muy útil para poder diseñar políticas de migración personalizadas.

4.10 Amoeba

Amoeba [TRS+90, MRT+90] es un sistema operativo distribuido basado en un micronúcleo [Gie90], desarrollado en la Universidad de Vrije y el Center for Mathematics and Computer Science de Amsterdam.

El diseño de Amoeba estuvo condicionado por los siguientes objetivos principales: transparencia, soporte para programación paralela, uso de la orientación a objetos y aproximación micronúcleo.

- **Transparencia**: se trata de proporcionar una imagen de sistema único. Para conseguir este objetivo, la decisión de diseño más importante fue la utilización del modelo del **conjunto** (*pool*) **de procesadores**, en el que no existe el concepto de máquina del usuario, y todos los recursos pertenecen al sistema como un todo.
- **Soporte para programación paralela**: aunque la transparencia es muy útil para casi todos los usuarios de un sistema distribuido, algunos de ellos están interesados en utilizarlo como plataforma de pruebas para experimentar con algoritmos, lenguajes, herramientas y aplicaciones distribuidas y paralelas. Amoeba proporciona soporte para este tipo de usuarios haciendo accesible para ellos el paralelismo subyacente.
- **Aproximación orientada a objetos, basada en capacidades**: otro de los objetivos fundamentales de Amoeba fue investigar la posibilidad de utilizar una aproximación orientada a objetos basada en capacidades para la construcción de un sistema distribuido. Con este motivo, los objetos y las capacidades se utilizan de una manera uniforme en el diseño del sistema operativo. En particular, el software del sistema operativo está basado en objetos, los cuales tienen nombre y están protegidos con el uso de capacidades.
- **Aproximación micronúcleo**: con este objetivo se intenta minimizar el tamaño del núcleo con el fin de incrementar la flexibilidad. De esta manera, muchos de los servicios estándar del sistema (por ejemplo, el servicio de ficheros) están construidos en el espacio de usuario, de tal forma que pueden ser fácilmente modificados.

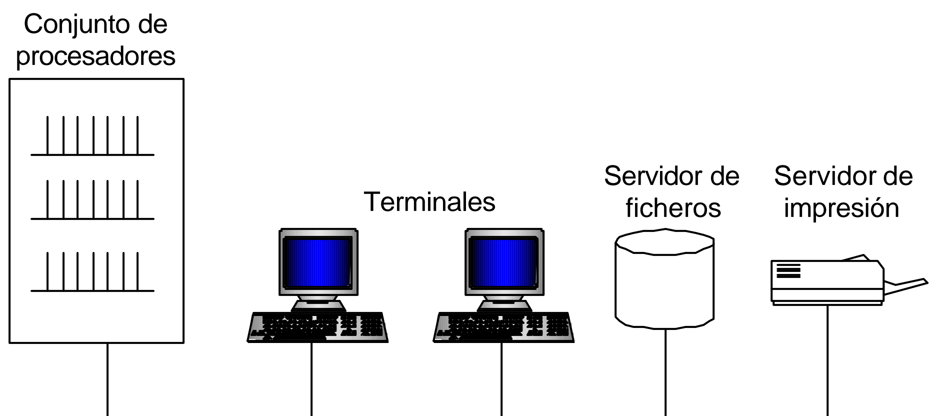


Figura 4.8. Arquitectura del sistema operativo Amoeba.

4.10.1 Objetos de Amoeba

Un objeto de Amoeba es similar a un tipo abstracto de datos, formado por un conjunto de datos encapsulados y un conjunto de operaciones que actúan sobre ellos. Los **objetos son pasivos** por naturaleza, de manera que no pueden hacer nada por sí mismos. Todo objeto es gestionado por un proceso servidor, que realiza las operaciones definidas por el objeto que gestiona y al que se pueden enviar solicitudes vía RPCs. Toda RPC especifica el objeto a utilizar (realmente se utiliza una capacidad, que se describe más adelante), la operación a realizar y los parámetros pasados y se ejecuta de manera síncrona. Ejemplos típicos de objetos Amoeba son los ficheros, directorios, segmentos de memoria, procesadores, discos, etc.

4.10.2 Capacidades

Todos los objetos están identificados y protegidos por una **capacidad**, que está compuesta por los siguientes campos:

- **Número de puerto del servidor**: identifica el servidor que gestiona el objeto referenciado por la capacidad.
- **Número de objeto**: dado que un servidor puede gestionar varios objetos del mismo tipo, el número de objeto permite al servidor identificar el objeto específico de entre todos los objetos que gestiona. La combinación del número de puerto del servidor y el número de objeto permite identificar unívocamente un objeto en el sistema.
- **Derechos**: denota qué operaciones puede realizar el poseedor de la capacidad sobre el objeto referenciado.
- **Control**: sirve para validar la capacidad, de tal manera que se evite la posibilidad de que se falsifiquen capacidades.

Las capacidades son gestionadas íntegramente por los procesos de usuario. Para crear un objeto, un proceso cliente envía un mensaje al servidor apropiado. El servidor crea el objeto y devuelve la capacidad al proceso cliente. Esta capacidad se denomina **capacidad del propietario**, y proporciona todos los derechos sobre el objeto. Si el propietario del objeto desea pasar derechos de acceso restringidos sobre el mismo a otros procesos, envía un mensaje al servidor solicitando la creación de una **capacidad restringida**, la cual, una vez creada, es enviada finalmente al proceso destinatario.

4.10.3 Nombrado de objetos

El **esquema de nombrado** basado en capacidades es transparente respecto a la localización del objeto y del servidor, dado que para realizar una operación sobre un objeto, no es necesario conocer la localización del objeto ni del servidor que lo gestiona. Para proporcionar dicha transparencia a los procesos de usuario, el núcleo es el encargado de resolver la localización de un objeto o servidor. El mecanismo de localización consiste en una serie de **preguntas difundidas** (*broadcast queries*) en toda la red, que incluyen el número del puerto del servidor que gestiona el objeto buscado. El mensaje de respuesta contiene la dirección de red en la que reside el servidor, que es almacenada en una cache para su futuro uso.

4.10.4 Crítica

Modelo de objetos como tipo abstracto de datos

A pesar de que en la terminología se hable de objetos, las entidades que utiliza Amoeba son más bien tipos abstractos de datos. No se soportan, por tanto, herencia, polimorfismo, etc.

Objetos pasivos

Los objetos se convierten en meros contenedores de datos y operaciones, recayendo la capacidad de computación en los procesos.

Implantación en el núcleo de mecanismos relacionados con la distribución

La introducción en el núcleo del mecanismo de localización de objetos dificulta notablemente su adaptación. No es posible cambiar el mecanismo o la política que lo rige sin modificar el propio núcleo del sistema operativo.

4.10.5 Características interesantes

Construcción del sistema operativo como conjunto de objetos

En Amoeba se consigue cierto grado de uniformidad, al estar construido el propio sistema operativo como conjunto de objetos. De esta manera, los objetos de usuario pueden invocar a otros objetos de usuario o al sistema operativo de una manera uniforme.

Transparencia de localización y acceso

Todos los objetos pueden ser accedidos de manera transparente a partir de una capacidad. El núcleo del sistema operativo se encargará de realizar las operaciones necesarias para proporcionar dicha transparencia.

Cache de localizaciones remotas de objetos

En otros sistemas operativos las pistas de localización eran almacenadas en las referencias a los objetos. En Amoeba se agrupan todas las pistas en una cache mantenida por el núcleo del sistema operativo.

4.11 Emerald

Emerald es un lenguaje de programación orientado a objetos para la construcción de aplicaciones distribuidas [Hut87, Jul88, RTL+91]. Aunque se desarrolló originalmente para simplificar la construcción de aplicaciones distribuidas eficientes, Emerald proporciona un modelo de programación de propósito general.

4.11.1 Objetos de Emerald

Un objeto de Emerald es el único mecanismo de abstracción del lenguaje. Consiste en:

- Un **nombre único** en todo el sistema distribuido.
- Su **representación**, es decir, los datos locales del objeto.
- Un **conjunto de operaciones** que pueden ser invocadas en el objeto.
- Un **proceso opcional**, que se arranca cuando el objeto es inicializado y se ejecutará en paralelo con las invocaciones a las operaciones del objeto.

Adicionalmente, un objeto puede tener varios atributos. Todo objeto tiene una **ubicación**, que especifica el nodo en el que reside actualmente el objeto. Un objeto puede ser definido como **inmutable**, definición que es dada por el programador y que indica que su estado no puede cambiar.

4.11.2 Actividad interna de los objetos

Emerald soporta concurrencia entre objetos y dentro de los objetos. Dentro de los objetos pueden estar progresando varias invocaciones en paralelo con el proceso interno del objeto.

4.11.3 Comunicación y localización

El único **mecanismo de comunicación** en Emerald es la invocación, siempre **síncrona**. En principio todos los argumentos y el resultado de una invocación se pasan por referencia. Existen, sin embargo, optimizaciones para permitir el paso de determinado tipo de argumentos por valor.

En general, los objetos en Emerald no tienen por qué estar al tanto de su **ubicación** para ejecutarse de manera adecuada, proporcionándose la **distribución de una manera transparente** al programador. Pero existe cierto tipo de aplicaciones (bases de datos distribuidas, servidores de nombres distribuidos, etc.) que existen porque existen entornos distribuidos. Para este tipo de aplicaciones el lenguaje proporciona primitivas que le permiten controlar la ubicación y el movimiento de los objetos.

4.11.4 Migración de objetos

Emerald proporciona un conjunto de primitivas para la distribución que permiten:

- Obtener la información de ubicación de un objeto.
- Mover un objeto a otro nodo.
- Fijar (atar) un objeto en un nodo y liberarlo de su atadura.

Adicionalmente, un objeto puede sufrir una modificación en su ubicación si se utiliza como parámetro en invocaciones remotas. Emerald proporciona tres variantes para el paso de parámetros por referencia, que es el modo por defecto y el más habitual:

- *Call-by-move*: el objeto representado por el parámetro se mueve al nodo en el que se encuentra el objeto invocado.
- *Call-by-visit*: muy parecido al anterior. Únicamente hay que añadir que el objeto parámetro vuelve a su nodo de origen una vez haya finalizado la invocación.
- *Call-by-move-return*: en este caso, el objeto resultado se mueve al nodo del objeto que realizó la invocación.

Es importante señalar que estas variantes en el paso de parámetros son conseguidas con la utilización de palabras clave del lenguaje que acompañan a los diferentes parámetros de una invocación.

En cualquiera de los casos anteriores, los atributos (estado) del objeto migran solidariamente con él.

4.11.5 Agrupación de objetos

Con el fin de optimizar el rendimiento en las operaciones de migración, Emerald proporciona primitivas para crear grupos de objetos, que se tienen en cuenta en las operaciones de migración. Se identifica un objeto como el principal, de tal forma que el resto de los objetos migrarán cuando lo haga el objeto principal.

4.11.6 Crítica

Uso restringido

El hecho de que Emerald sea un lenguaje de programación con características distribuidas limita su uso a aquellas plataformas para las cuales existe soporte.

Además, algunas primitivas de distribución (por ejemplo, migración en el paso de parámetros) son proporcionadas a nivel sintáctico, y no es posible modificarlas o adaptarlas.

Modelo de objetos incompleto

Una característica que se ha descrito como imprescindible en un modelo de objetos, como es la herencia, no está presente en Emerald. De nuevo los objetos se convierten en

tipos abstractos de datos. La inexistencia de herencia en el modelo simplifica notablemente los distintos mecanismos de la distribución.

4.11.7 Características interesantes

Soporte de objetos de cualquier granularidad

Los objetos soportados por Emerald pueden ser de cualquier granularidad. El objeto se convierte en la única entidad que maneja el sistema, proporcionándose la deseada uniformidad conceptual.

Transparencia de localización y acceso

Los objetos son accedidos de manera transparente con respecto a su ubicación. En este sentido, el nombre único asociado a todos los objetos permite localizar un objeto de manera unívoca dentro del sistema distribuido.

Control de la distribución

El programador puede ejercer el control sobre la distribución de los objetos haciendo uso de un conjunto de primitivas al efecto. Esto es muy útil para determinadas aplicaciones que pueden necesitar sacar partido de la arquitectura distribuida del sistema.

Agrupaciones de objetos

La agrupación de objetos permite minimizar el número de invocaciones remotas generadas por la migración de un objeto. El programador será el responsable de determinar qué objetos tienen que moverse de manera solidaria, aunque determinadas agrupaciones pueden ser establecidas por el propio sistema (por ejemplo, los atributos se mueven con el propio objeto).

4.12 COOLv2

COOLv2 [LJP93] es una capa de soporte para objetos distribuidos construida sobre el micronúcleo CHORUS [RAA+92]. El proyecto se proponía reducir la desadaptación de impedancias entre las abstracciones de los lenguajes y las abstracciones proporcionadas por el sistema. Para ello extiende el micronúcleo CHORUS con abstracciones más adecuadas para sistemas orientados a objeto, con la idea de reducir la ineficiencia de capas software añadidas.

4.12.1 Abstracciones base

Las abstracciones incluidas en el sistema base son los **agrupamientos** (***cluster***) y los **espacios de contexto** que abstraen los micronúcleos distribuidos y el almacenamiento persistente.

Un agrupamiento es un conjunto de regiones de memoria respaldadas en disco que serán usadas para colocar sobre ellas los objetos. Los agrupamientos se hacen corresponder con espacios de direcciones virtuales distribuidos que forman un espacio de contexto.

4.12.2 Soporte genérico en tiempo de ejecución

Sobre las abstracciones base se coloca una capa de software denominada **GRT** (*Generic Run Time*, **soporte genérico en tiempo de ejecución**). El GRT implementa la noción de objetos usando un modelo de objetos básicos organizados en clases. También

proporciona invocación de métodos y actividades (hilos) e interactúa con los niveles inferiores para valerse del soporte que proporcionan para la persistencia y la memoria compartida distribuida.

4.12.3 Soportes específicos para lenguajes

Este GRT se complementa con **soportes en tiempo de ejecución específicos** para diferentes lenguajes (C++, Eiffel, etc.). La combinación del GRT con el soporte específico de un lenguaje proporciona el soporte para el modelo de objetos de un lenguaje determinado.

4.12.4 Modelo de objetos

Los objetos se tratan como objetos pasivos. Los hilos de ejecución (**actividades**) viajan, por tanto, de objeto en objeto mediante las invocaciones de métodos.

Todo objeto existe dentro de un **contexto**, que le sirve como entorno de ejecución.

Un conjunto de atributos asociados a un objeto determina si es conocido globalmente y si es persistente. Un objeto tiene que solicitar la recepción de mensajes enviados por objetos remotos, convirtiéndose en un objeto global.

4.12.5 Invocación de objetos

Existen dos maneras de invocar los objetos. Dentro de un agrupamiento se accede a los objetos locales utilizando las referencias propias de cada lenguaje, que serán direcciones de memoria virtual (punteros). Para invocar a objetos que no están en el agrupamiento, se utiliza un **objeto de interfaz** (*proxy* o representante) que representa al objeto remoto y al que se accede usando un identificador global persistente. El código para estos representantes es generado por compiladores especiales modificados para ello.

4.12.6 Distribución

Los objetos se identifican de manera transparente con respecto a su localización, lo que hace a su vez transparente su migración. De la misma forma pueden migrar del almacenamiento secundario a un contexto cuando no se encuentran en el almacenamiento volátil.

COOL está organizado como un **conjunto de servidores**, en cada uno de los cuales existe un gestor (*manager*) que proporciona la gestión básica de objetos y contextos del servidor. Los servidores COOL se comunican utilizando los mecanismos IPC del micronúcleo Chorus subyacente.

Los objetos creados con el **atributo global** pueden ser el objetivo de invocaciones remotas, pero tienen que solicitarlo de manera explícita. Las invocaciones no son tales en el sentido estricto de la palabra, sino que son, más bien, operaciones de paso de mensajes.

La **migración de objetos** se produce en las operaciones de paso de mensaje, de tal forma que el objeto emisor especifica los objetos a ser movidos o copiados en el contexto destino. El soporte en tiempo de ejecución para cada lenguaje pueden construir sobre el mecanismo elemental primitivas de migración o semánticas de paso de parámetros como las descritas para Emerald. En cualquier caso, lo que realmente migran son los identificadores de los objetos. Los objetos propiamente dichos serán reclamados en el destino a través de los mecanismos de memoria virtual subyacentes.

4.12.7 Crítica

Falta de uniformidad en la orientación a objetos

El uso de objetos no tiene una uniformidad total. Por ejemplo se usan dos maneras de referenciar a los objetos, en función de su situación. Dentro del mismo módulo se usan las referencias de cada lenguaje, y para acceder a un objeto remoto se utiliza un mecanismo de acceso diferente basado en un identificador global.

Los objetos deben solicitar explícitamente poder ser conocidos globalmente

Por defecto, los objetos no pueden ser invocados por otros objetos. Deben solicitarlo explícitamente. Este hecho resta cierta transparencia al mecanismo de invocación.

Paso de mensajes en lugar de invocación

La invocación de objetos no existe como tal. Es remplazada por el mecanismo de paso de mensajes proporcionado por el micronúcleo Chorus.

Orientación a objetos sólo en el espacio del usuario: pérdida de flexibilidad en el sistema

El soporte para objetos sólo existe en las aplicaciones de usuario. El resto del sistema es convencional y por tanto hay que acceder mediante las interfaces no orientadas a objetos del mismo. Las ventajas de la orientación a objetos para la extensibilidad, etc. no se pueden aplicar al sistema base. Se pierde flexibilidad.

4.12.8 Características interesantes

Mecanismo mínimo de migración

El establecimiento de un mecanismo mínimo de migración de objetos no sujeto a ningún tipo de política permite que el sistema sea flexible, en el sentido que otras entidades de más alto nivel serán las que tengan que determinar de qué manera va a tener lugar la migración. Es posible, por tanto, que diferentes lenguajes de programación utilicen políticas o semánticas distintas sobre el mismo mecanismo básico.

4.13 APERTOS

Apertos [Yok92], una evolución de su predecesor Muse [YTY+89] suele considerarse como el pionero en la aplicación de la **reflectividad** en los sistemas operativos orientados a objetos.

Es un sistema operativo para dar soporte a objetos y estructurado uniformemente en términos de objetos. La motivación inicial de aplicación es para su uso en un entorno de computación móvil, compuesto por ordenadores que pueden estar conectados a la red y también desconectarse para trabajar independientemente o trabajar remotamente mediante enlaces sin hilos, etc. La gran novedad de este sistema consiste en utilizar una separación de los objetos en dos niveles: meta-objetos y objetos base.

4.13.1 Estructuración mediante objetos y meta-objetos

Los **meta-objetos** proporcionan el entorno de ejecución y dan soporte (definen) a los **objetos base**. Cada objeto base está soportado por un **meta-espacio**, que está formado por un grupo de meta-objetos. Cada meta-objeto proporciona una determinada funcionalidad a los objetos del meta-espacio en que se encuentra. Por ejemplo, un determinado objeto puede estar en un meta-espacio con meta-objetos que le

proporcionen una determinada política de planificación, un mecanismo de sincronización determinado, el uso de un protocolo de comunicación, etc.

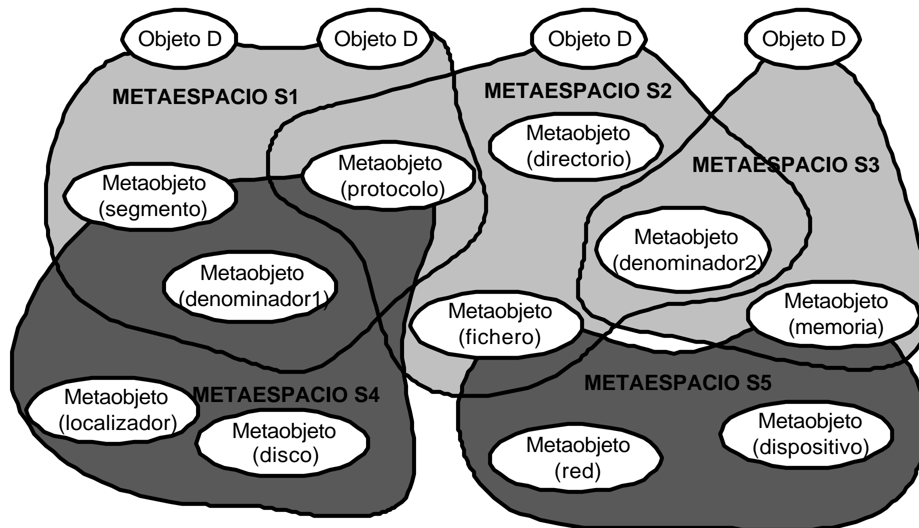


Figura 4.9. Separación entre los espacios de objetos y de meta-objetos en el sistema Apertos.

4.13.2 Flexibilidad

La flexibilidad del sistema se consigue mediante la posibilidad de que un objeto cambie de meta-espacio, o la adición de nuevos meta-objetos que proporcionen funcionalidad adicional. Por ejemplo, un meta-objeto podría migrar de un meta-espacio dado a otro meta-espacio que use un protocolo de comunicaciones para una red sin hilos cuando el ordenador en el que se encuentre se desconecte físicamente de la red.

4.13.3 Reflectividad

La reflectividad se produce al existir una interfaz bidireccional entre los objetos base y los meta-objetos que los soportan. Los objetos pueden dialogar con sus meta-objetos, usando un punto de entrada al meta-espacio denominado **reflector**. A su vez, los meta-objetos influyen en el funcionamiento de los objetos base. Por otro lado ambos, objetos y meta-objetos, comparten el mismo marco conceptual al estar descritos en los mismos términos de objetos. Los meta-objetos también pueden considerarse como objetos, por lo que tendrían su propio meta-meta-espacio y así sucesivamente. Esta regresión termina en un meta-objeto primitivo que no tiene meta-meta-espacio (se describe a sí mismo).

Los objetos y meta-objetos se organizan en una jerarquía de clases, al igual que los reflectores. Apertos se estructura usando una serie de jerarquías predefinidas de reflectores y meta-objetos.

4.13.4 Jerarquía de reflectores

La jerarquía de reflectores define la estructura común de programación de meta-objetos del sistema. Un reflector, como punto de entrada a un meta-espacio, ofrece a los objetos base una serie de operaciones que pueden usar de sus meta-objetos. La jerarquía base define las operaciones comunes del sistema.

MetaCore es un meta-objeto terminal en el que finaliza la regresión de meta-meta-objetos. Puede considerarse como el equivalente de un micronúcleo de otros sistemas. Proporciona a todos los demás objetos del sistema con los elementos básicos del sistema, es decir, las primitivas comunes de separación entre objetos y meta-objetos y de migración de objetos. Proporciona el concepto de contexto de ejecución para virtualizar la CPU y las primitivas básicas de la computación reflectiva.

4.13.5 Invocación de objetos

Los objetos del nivel base pueden comunicarse entre sí de manera uniforme sin necesidad de tener en cuenta si el objeto invocado reside o no en la misma máquina. La naturaleza de la comunicación (local o remota) es determinada por un meta-objeto, que llevará a cabo las acciones oportunas para que tenga lugar la invocación.

4.13.6 Migración de objetos

El concepto de migración de objetos en Apertos es ligeramente distinto del utilizado hasta ahora. En Apertos, un objeto migra cuando cambia su metaespacio, es decir, el objeto atraviesa una metajerarquía.

La migración es realiza por metaobjetos, tanto en el metaespacio origen como en el metaespacio destino. Dado que un objeto es representado internamente como un conjunto de metaobjetos, la migración de un objeto consiste en la migración de metaobjetos al metaespacio destino.

4.13.7 Crítica

Complejidad de estructura

La separación de la estructura del sistema en múltiples meta-espacios recursivos, aunque da mucha flexibilidad al sistema, también complica la comprensión del mismo de manera sencilla por el usuario.

Falta de uniformidad por la separación espacio/meta-espacio de objetos

La separación completa de ambos espacios introduce una cierta falta de uniformidad en el sistema. La jerarquía de meta-objetos está totalmente separada de la de los objetos de usuario. La programación de meta-objetos se realiza por tanto de una manera diferente a la de los objetos normales.

4.13.8 Características interesantes

Reflectividad para la flexibilidad

La adopción de la reflectividad en el sistema es muy importante para lograr la flexibilidad en el sistema de manera uniforme. Esta propiedad permite describir mediante objetos los propios elementos de la máquina. De esta manera se unifican dentro del paradigma de la OO los elementos del usuario y los elementos del sistema que los soportan. Esto permite la extensibilidad del sistema de una manera uniforme, al permitir que los objetos de usuario puedan acceder a los objetos del sistema usando el mismo paradigma de la OO.

Migración entre metaespacios

La migración de objetos entre metaespacios permite extender el concepto habitual de migración (entre diferentes nodos o computadores). Un objeto puede migrar, por

ejemplo, a un almacenamiento secundario, siempre que esté representado por el metaespacio correspondiente.

CAPÍTULO 5 NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS

5.1 Introducción

Tal y como se ha visto en el capítulo anterior, la orientación a objetos se introduce, en general, de una manera poco uniforme y completa en la construcción de sistemas operativos orientados a objetos. En la mayoría de los casos, es debido a que los sistemas operativos siguen fundamentándose en la abstracción tradicional de proceso, separada de los datos sobre los que trabajan, y proporcionando a los programas de usuario el concepto de objeto, generándose el problema de la **desadaptación de impedancias** o salto semántico (entre paradigma procedimental y orientado a objetos). Algún otro sistema ofrece de manera uniforme el soporte de objetos, pero se queda demasiado lejos de lo que comúnmente se acepta como características del modelo de objetos.

5.2 Necesidad de un sistema integral orientado a objetos

Una manera de resolver los problemas mencionados en la introducción es **crear un sistema homogéneo** en el que se utilice en todos sus elementos el mismo paradigma de la orientación a objetos y se dé soporte nativo a los mismos: un **sistema integral orientado a objetos**.

En un sistema integral como éste se crea un entorno de computación en el que todos los elementos: lenguajes, aplicaciones, compiladores, interfaces gráficas, bases de datos, etc. hasta los más cercanos a la máquina comparten el mismo paradigma de la orientación a objetos. El sistema comprende el concepto de objeto y es capaz de gestionar directamente objetos.

Al usar el mismo paradigma **no existe desadaptación**. Desde una aplicación OO tanto el acceso a los servicios del sistema como la interacción con otros objetos se realizan utilizando los mismos términos OO.

Al dar soporte directo a objetos, es decir, al gestionar directamente tanto la representación de los objetos como su utilización se soluciona el problema de la interoperabilidad. Los objetos no son conceptos que sólo existen dentro de los procesos del SO y que sólo son conocidos por el compilador que creó el proceso. Al ser gestionados por el propio sistema pueden "verse" unos a otros, independientemente del lenguaje y del compilador que se usó para crearlos.

No son necesarias capas de adaptación con lo que se reduce la complejidad conceptual y técnica del sistema. Los usuarios/programadores se ven liberados de preocuparse por detalles técnicos y contraproductivos cambios de paradigma y pueden concentrarse en el aspecto más importante que es la resolución de los problemas usando la OO.

5.3 Características del sistema integral orientado a objetos

La idea de crear un sistema que dé soporte directo y utilice exclusivamente el paradigma de la orientación a objetos nos conduce a la siguiente definición de sistema integral orientado a objetos:

Un sistema integral orientado a objetos ofrece al usuario un entorno de computación que crea un mundo de objetos: un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados.

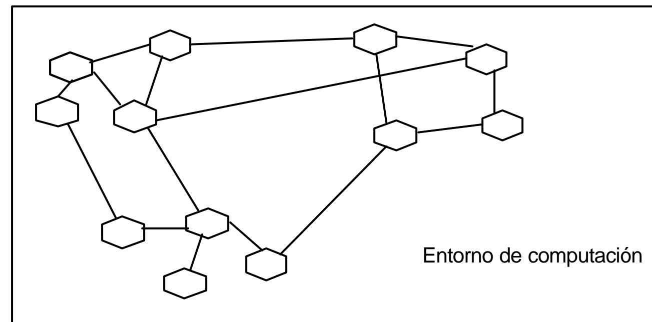


Figura 5.1. Entorno de computación compuesto por un conjunto de objetos homogéneos.

Este sistema permitirá aprovechar al máximo las conocidas ventajas de la orientación a objetos [Boo94]: reutilización de código, mejora de la portabilidad, mantenimiento más sencillo, extensión incremental, etc. en todos los elementos del sistema y no solo en cada aplicación OO como en los sistemas convencionales.

Los requisitos que deben estar presentes en un entorno como éste son los siguientes:

- Uniformidad conceptual en torno a la orientación a objetos
- Transparencia: persistencia y **distribución**
- Heterogeneidad y portabilidad
- Seguridad
- Concurrencia
- Multilenguaje / Interoperabilidad
- Flexibilidad

5.3.1 Uniformidad conceptual en torno a la orientación a objetos

El único elemento conceptual que debe utilizar el sistema es un mismo paradigma de orientación a objetos. El sistema proporcionará una única perspectiva a los usuarios/programadores: la de objetos, que permite comprender más fácilmente sistemas cada vez más complicados y con más funcionalidad [YMF91].

Modo de trabajo exclusivamente orientado a objetos

La única abstracción es, por tanto, el objeto (de cualquier granularidad), que encapsula toda su semántica. Lo único que puede hacer un objeto es crear nuevas clases

que hereden de otras, crear objetos de una clase y enviar mensajes a otros objetos. Toda la semántica de un objeto se encuentra encapsulada dentro del mismo.

Homogeneidad de objetos

Todos los objetos tienen la misma categoría. No existen objetos especiales. Los propios objetos que den soporte al sistema no deben ser diferentes del resto de los objetos.

Esta simplicidad conceptual hace que todo el sistema en su conjunto sea fácil de entender, y se elimine la desadaptación de impedancias al trabajar con un único paradigma. La tradicional distinción entre los diferentes elementos de un sistema: hardware, sistema operativo y aplicaciones de usuario se difumina.

5.3.2 Transparencia

El sistema debe hacer transparente la utilización de los recursos del entorno al usuario y en general todas las características del sistema, en especial:

Distribución

El sistema debe ser inherentemente distribuido, ocultando al usuario los detalles de la existencia de numerosas máquinas dentro de una red, pero permitiendo la utilización de las mismas de manera transparente en el entorno de computación. Con términos de objetos, los objetos podrán residir en cualquier máquina del sistema y ser utilizados de manera transparente independientemente de cual sea esta.

Persistencia

El usuario no debe de preocuparse de almacenar los objetos en memoria secundaria explícitamente. El sistema se debe de ocupar de que el usuario perciba un único espacio de objetos y transparentemente almacenarlos y recuperarlos de la memoria secundaria.

5.3.3 Heterogeneidad y portabilidad

El sistema no debe obligar a la utilización de un determinado modelo de máquina para su funcionamiento. Debe tenerse en cuenta la existencia de numerosos tipos de máquinas dentro de la misma red de trabajo de una organización, que posiblemente sean incompatibles entre sí.

Por la misma razón, para llegar al mayor número de máquinas posible, interesa que el esfuerzo para portar el propio sistema de una máquina a otra sea el menor posible.

5.3.4 Seguridad

El entorno de computación debe ser seguro y protegerse frente a ataques maliciosos o errores lógicos. El sistema dispondrá de un mecanismo de protección que permita controlar el acceso no autorizado a los objetos. Por supuesto, este mecanismo debe integrarse de manera uniforme dentro del paradigma OO.

5.3.5 Concurrencia

Un sistema moderno debe presentar un modelo de concurrencia que permita la utilización de los objetos aprovechando el paralelismo. Dentro de una misma máquina aprovechando la concurrencia aparente y en sistemas distribuidos o multiprocesador la concurrencia real.

5.3.6 Multilenguaje / Interoperabilidad

El sistema no debe restringir su utilización a sólo un lenguaje de programación. De esta manera la mayoría de los programadores no necesitarán aprender otro lenguaje. Además, algunos problemas se resuelven mejor en un determinado lenguaje que en otros.

Sin embargo, la interoperabilidad entre los diferentes lenguajes debe quedar asegurada, para evitar los problemas de desadaptación anteriores.

5.3.7 Flexibilidad

La **flexibilidad** se define como la capacidad de diseñar un sistema que ofrezca la posibilidad de ser personalizado para cumplir los requisitos específicos de aplicaciones o dominios de aplicación [Cah96a].

Diseñar un sistema operativo flexible supone que sea fácilmente personalizable para proporcionar a cada aplicación aquellas características que requiera, es decir, permitir a cada aplicación o conjunto de aplicaciones, configurar el sistema operativo de la forma más adecuada. De hecho, dado que algunas aplicaciones pueden funcionar de forma poco eficiente con la implantación concreta que el sistema operativo ofrezca por omisión, la posibilidad de personalizar una característica trae consigo el aumento del rendimiento.

Para un sistema experimental y de investigación como éste, la flexibilidad es muy importante. El sistema debe ser fácil de adaptar a entornos diferentes, como por ejemplo sistemas empotrados, sistemas sin disco duro, sistemas multiprocesador. También a los requisitos de las aplicaciones: algunas no necesitarán persistencia, otras una forma especial de la misma, etc. Muy a menudo se debe experimentar reemplazando o añadiendo nuevos servicios para comprobar el comportamiento del sistema, etc. En resumen, el sistema debe permitir eliminar, añadir o modificar funcionalidad de manera sencilla.

5.3.7.1 Causas y consecuencias de la falta de flexibilidad

Las causas fundamentales de la falta de flexibilidad en los sistemas operativos tradicionales son, en primer lugar, la ocultación de los detalles de implantación de una determinada característica; en segundo lugar, el carácter general e inamovible de los mecanismos y abstracciones que proporcionan y, en tercer lugar, la ausencia de algunas características requeridas por las aplicaciones.

Todo ello provoca que las aplicaciones tengan, en muchos casos, un pobre rendimiento, dado que los servicios que proporciona el sistema operativo no están implementados de forma óptima para cada una de las aplicaciones. Además, debido al comportamiento de caja negra de los sistemas operativos, no es posible explotar el conocimiento que tienen las aplicaciones acerca de cómo implementar las distintas características del sistema operativo.

5.3.7.2 Aproximaciones a la flexibilidad

Básicamente, existen dos aproximaciones para conseguir un sistema flexible: **flexibilidad estática y dinámica** [Cah96a].

Flexibilidad estática: sistemas personalizables

La flexibilidad estática permite que el sistema sea personalizado para una aplicación, conjunto de aplicaciones o bien para alguna configuración hardware particular **en tiempo de compilación, enlace o carga**.

Para lograr un sistema estáticamente flexible, se requiere poder implementar de diversas formas la arquitectura del sistema con el fin de incluir los mecanismos y las políticas requeridos por las aplicaciones.

Como ejemplos de tales sistemas están Choices [CIM92] y PEACE [SPr93b].

Flexibilidad dinámica

Supone que el sistema puede ser personalizado para cubrir o adaptarse a las necesidades de las aplicaciones **en tiempo de ejecución**. Para conseguir la flexibilidad dinámica existen diversos grados que se detallan a continuación.

- **Sistemas adaptables**. Un sistema adaptable se construye como un sistema de propósito general, posiblemente basado en las mismas abstracciones y mecanismos, pero proporciona servicios que soportan un amplio rango de políticas que dan cobertura a los requisitos del mayor número posible de aplicaciones [GIL95]. Sin embargo, su principal característica es que permite la personalización para cambiar dinámicamente durante la ejecución del programa de tal forma que pueda adecuarse dinámicamente a los requerimientos de las aplicaciones. Un ejemplo de tal tipo de sistemas es el micronúcleo Mach que permite a las aplicaciones dar pistas al sistema de planificación [Bla90].
- **Sistemas modificables.** Un paso más es permitir a la aplicación, no ya escoger entre un amplio rango de políticas ofrecidas, sino participar de alguna forma en la implantación de un servicio. Esta aproximación se consigue obligando al servicio del núcleo a efectuar una llamada ascendente a un módulo proporcionado por la aplicación que implementa alguna política para el servicio. Otra posibilidad es permitir que la aplicación interponga código a la interfaz del servicio. Un ejemplo de un sistema que soporta esta clase de flexibilidad es el micronúcleo Mach vía la interfaz con el paginador externo [YTR+87].
- **Sistemas configurables y extensibles**. Los sistemas configurables permiten a las aplicaciones reemplazar en tiempo de ejecución servicios del sistema para dar soporte a funcionalidad de diferente modo, para una aplicación particular. Si además permiten añadir nuevos servicios en tiempo de ejecución con el fin de proporcionar funcionalidad nueva, tales sistemas reciben el nombre de **extensibles**. La mayoría de los micronúcleos son configurables dado que por su propia arquitectura tienen muchos módulos del sistema operativo implementados como servidores en el espacio del usuario por lo que pueden ser instalados o reemplazados en tiempo de ejecución. Como ejemplo de sistema configurable está el SPIN [BSP+95] que permite incluso instalar servicios dinámicamente en el núcleo.

5.4 Modelo de objetos del sistema integral

El modelo de objetos del sistema integral sigue las líneas básicas descritas en el capítulo 3 para cualquier sistema de distribución de objetos.

Se trata de un modelo de objetos intencionadamente estándar, que recoge las características más utilizadas en los lenguajes de programación más populares y, en general, más aceptadas entre la comunidad OO:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía de herencia y agregación
- Tipos
- Concurrencia
- Persistencia
- **Distribución**

5.5 Resumen

La construcción de un sistema operativo distribuido orientado a objetos con las características descritas en el capítulo 3 no es trivial. La revisión de un conjunto de sistemas operativos en el capítulo anterior arroja como resultado que la orientación a objetos no se aplica de manera completa en su diseño y construcción, sustentándose ambas de una manera importante en abstracciones tradicionales como la de proceso.

Con el fin de construir un sistema operativo distribuido orientado a objetos en el que el objeto se convierta en la abstracción única, es necesario que el soporte de objetos se proporcione desde las capas más bajas del mismo, y de una manera homogénea para todo el sistema: sistema operativo y programas de usuario. Surge entonces la necesidad de construir un sistema integral orientado a objetos, en el que, entre otras cosas, se construirá un sistema operativo orientado a objetos con la funcionalidad deseada.

Las características del sistema integral deben sintonizar con las descritas para un sistema operativo distribuido orientado a objetos. Dado que en ambos casos el objeto es la entidad única para su construcción, se trata de que el sistema integral adopte el modelo de objetos especificado para los sistemas distribuidos orientados a objetos, de manera que se mantenga la uniformidad en todo el sistema.

CAPÍTULO 6 ARQUITECTURA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS

6.1 Introducción

Este capítulo describe una arquitectura que permite obtener un sistema integral orientado a objetos para proporcionar soporte para un sistema operativo distribuido orientado a objetos.

La arquitectura a adoptar para la construcción del sistema integral debe permitir mantener la uniformidad conceptual en torno a la orientación a objetos, pero con el objetivo último de integrar de manera flexible la distribución de objetos en el sistema.

6.2 ¿Cómo ofrecer la funcionalidad de la distribución de objetos?

El sistema integral proporciona el soporte de objetos desde las capas más profundas, en lo que se podría denominar el **núcleo del sistema integral**. El núcleo del sistema integral debe proporcionar la funcionalidad básica del modelo de objetos, con el fin de obtener las ventajas de una aproximación micronúcleo como la descrita en el capítulo 2. Por otro lado, existe un conjunto de características adicionales, como la persistencia y la distribución, que tienen que ser proporcionadas en el sistema integral.

Existen fundamentalmente tres posibilidades no excluyentes entre sí para completar el modelo básico de objetos proporcionado por el núcleo del sistema integral con el resto de características mencionadas, de forma que el resultado sea un entorno de computación distribuido OO: implementación en el propio núcleo de soporte de objetos, introducción de clases y objetos especiales y extensión no intrusiva del núcleo básico.

6.2.1 Implementación ad-hoc en el núcleo básico de soporte de objetos

Una posibilidad para añadir una característica al sistema integral, como puede ser la introducción de la distribución en el modelo, sería la **modificación del núcleo básico** de soporte de objetos.

El modelo de objetos del núcleo del sistema integral englobaría todas las características mencionadas, de tal forma que el sistema integral proporcionase obligatoriamente soporte para objetos distribuidos.

Este enfoque presenta varios problemas. En primer lugar, al integrarse todas las características en el núcleo, éste tiende a crecer en tamaño y complejidad, perdiéndose las ventajas de una aproximación micronúcleo para la construcción del sistema integral. En segundo lugar, el sistema sería difícilmente adaptable, dado que sería necesario modificar el núcleo para configurar el sistema integral de acuerdo a unas necesidades concretas. Por ejemplo, la gestión de nuevos protocolos de red para la implantación del sistema integral en nuevos entornos, obligaría la modificación del núcleo. Por último, y por la misma razón anterior, sería difícil y costoso extender el sistema.

6.2.2 Introducción de clases y objetos especiales que proporcionen la distribución

Todos los lenguajes de programación proporcionan un conjunto de tipos de datos básicos y unas reglas para construir tipos de datos definidos por el usuario. En los lenguajes OO ocurre lo mismo, pero con clases básicas y clases definidas por el usuario. El núcleo del sistema integral proporcionará un conjunto de clases básicas o primitivas.

Cualquier aplicación se construye basándose, no sólo en las primitivas proporcionadas por el núcleo, sino utilizando las clases que ofrece, instanciando objetos de dichas clases, especializándolas o añadiendo nuevas clases.

Para añadir nuevas características o modificar el comportamiento del SIOO, una posibilidad, además de la anterior, podría ser la modificación de la funcionalidad de estas clases mediante la adición de nuevos atributos y/o métodos lo que conlleva un cambio en el funcionamiento de la máquina de cara al exterior o la modificación de su comportamiento.

Un ejemplo es la posibilidad de modificar la semántica del objeto para ofrecer un modelo de objetos activo. La máquina abstracta tiene como raíz de la jerarquía que expone al exterior una clase raíz, clase de la que cualquier otra clase, definida en la máquina o por el usuario, debe heredar, heredando por tanto, la semántica definida por ella.

Si esa clase se implanta dotando al objeto de características relacionadas con la computación, tal como se vio en el capítulo 3, la máquina abstracta da soporte a una abstracción de objeto activo y cualquier objeto que se cree en el sistema también seguirá este modelo, lográndose la ansiada homogeneidad.

6.2.3 Extensión no intrusiva del núcleo básico

Una tercera posibilidad es utilizar el núcleo para ofrecer el soporte básico a la existencia de objetos, así como los mecanismos básicos para construir un entorno de ejecución para los mismos, a la vez que se permite que sus mecanismos puedan ser configurados por objetos no diferentes de cualquier objeto de usuario normal.

Así, estos objetos modifican la funcionalidad del núcleo, **extendiendo su comportamiento**.

Para ello, es necesario que el núcleo esté construido de tal forma que permita que un objeto externo sustituya a un objeto propio para ofrecer la misma funcionalidad de una forma distinta, manteniendo de esta forma la **uniformidad en torno a la OO.**

El problema es cómo realizar la comunicación entre el núcleo y los objetos de usuario sin romper la uniformidad en la OO. La solución es dotar al núcleo de una **arquitectura reflectiva OO** [Mae87].

La reflectividad hace que el núcleo sea expuesto a los objetos de usuario en forma de un conjunto de objetos normales, de manera que no se diferencien unos y otros. Así, los objetos del núcleo pueden usar los objetos normales, y viceversa, usando el mismo marco de la OO común en el sistema.

El sistema se puede ver como un **conjunto de objetos homogéneos** que interaccionan entre sí, compartiendo las mismas características (el mismo modelo de objetos). Algunos de ellos serán objetos del núcleo, otros objetos normales de usuario. De estos últimos, algunos implementarán funcionalidad del sistema operativo. Sin embargo, el origen de un objeto es transparente para todos los demás y para sí mismo.

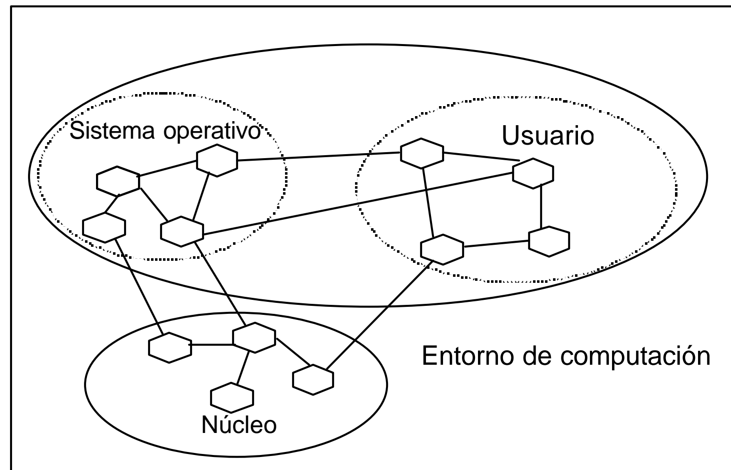


Figura 6.1. Espacio de objetos homogéneo formado por la unificación de los objetos del núcleo con los del sistema operativo y los del usuario.

La reflectividad está reconocida como una técnica importante para lograr flexibilidad (extensibilidad, adaptabilidad, etc.) en los núcleos de los sistemas operativos [Cah96a, GW96]. Cuando se utiliza en un sistema integral orientado a objetos contribuye a la uniformidad, como se puede ver. Debido a la importancia de la flexibilidad para el sistema integral el siguiente capítulo se dedica a un estudio más profundo de la reflectividad para hacer una selección de qué tipo de reflectividad y qué alternativas existen para implantarla en el núcleo del sistema integral.

6.3 Sistema Integral OO = Maquina abstracta OO + Sistema Operativo OO

A continuación se describe una arquitectura que permite obtener un sistema integral orientado a objetos con los objetivos especificados en el capítulo anterior.

La **arquitectura propuesta sigue la aproximación no intrusiva** descrita anteriormente, y consiste en una máquina abstracta orientada a objetos y un sistema operativo orientado a objetos. La máquina abstracta es el núcleo básico, que proporciona soporte para un modelo único de objetos de todo el sistema. Una serie de objetos que forman el sistema operativo extienden la funcionalidad de la máquina, proporcionando a los objetos diferentes facilidades (entre ellas, la distribución) de manera transparente.

Esta arquitectura [ATA+97] integra diferentes tendencias en el campo de los sistemas operativos avanzados, especialmente con características de OO, mostradas en el capítulo 4. En general, estas tendencias se han aplicado para solucionar problemas concretos en un sistema operativo determinado. Sin embargo, la arquitectura de este sistema permite aplicar estas tendencias de manera natural, conjunta y uniforme para obtener un sistema integral con las propiedades deseadas. La característica más importante es la integración fluida de las mismas dentro del marco general de la OO sin romper el objetivo fundamental del sistema.

6.3.1 Propiedades fundamentales de la arquitectura

No todas las características del modelo de objetos tienen que estar implementadas por el mismo elemento del sistema integral. De hecho, algunas de ellas, como la

persistencia e incluso la distribución, es más interesante introducirlas de manera que (aunque transparente) su uso pueda ser opcional (como se verá).

Las propiedades más importantes que incorpora la arquitectura anterior en cada uno de sus dos elementos, la máquina abstracta y el sistema operativo, se reseñan a continuación.

6.3.1.1 Máquina abstracta orientada a objetos

Dotada de una arquitectura reflectiva, proporciona un modelo único de objetos para el sistema. Se encargará de proporcionar las características fundamentales de este modelo de objetos, especialmente las más cercanas a los elementos normalmente encontrados en los lenguajes orientados a objetos.

Modelo único de objetos

La máquina proporciona al resto del sistema el soporte para objetos necesario. Los objetos se estructuran usando el modelo de objetos de la máquina, que será el único modelo de objetos que se utilice dentro del sistema.

- Identidad única de objetos
- Clases
- Herencia múltiple (es-un)
- Agregación (es-parte-de)
- Relaciones generales de asociación (asociado-con)
- Comprobación de tipos, incluso en tiempo de ejecución
- Polimorfismo
- Excepciones

Reflectividad

La máquina dispondrá de una arquitectura reflectiva [Mae87], que permita que los propios objetos constituyentes de la máquina puedan usarse dentro del sistema como cualquier otro objeto dentro del mismo. La arquitectura reflectiva concreta de la máquina abstracta se describe en el capítulo 9.

6.3.1.2 Sistema Operativo Orientado a Objetos

Estará formado por un conjunto de objetos que se encargarán de conseguir las propiedades que más comúnmente se asocian con funcionalidad propia de los sistemas operativos.

Transparencia: persistencia, distribución, concurrencia y seguridad

Estos objetos serán objetos normales del sistema, aunque proporcionarán de manera transparente al resto de los objetos las propiedades de **persistencia**, **distribución**, **concurrencia** y **seguridad**.

Hay que reseñar que bajo el epígrafe de "sistema operativo" se agrupa todo lo referente a la manera de conseguir esta funcionalidad. En general, esta se facilitará mediante un conjunto de objetos normales que extiendan la máquina abstracta para proporcionar a todos los objetos del sistema estas propiedades de manera transparente.

6.3.1.3 Orientación a objetos

Se utiliza en el sistema operativo al organizarse sus objetos en una jerarquía de clases, lo que permite la reusabilidad, extensibilidad, etc. del sistema operativo.

La propia estructura interna de la máquina abstracta también se describirá mediante la OO.

6.3.1.4 Espacio único de objetos sin separación usuario/sistema

La combinación de la máquina abstracta con el sistema operativo produce un único espacio de objetos en el que residen los objetos. No existe una división entre los objetos del sistema y los del usuario. Todos están al mismo nivel, independientemente de que se puedan considerar objetos de aplicaciones normales de usuario u objetos que proporcionen funcionalidad del sistema.

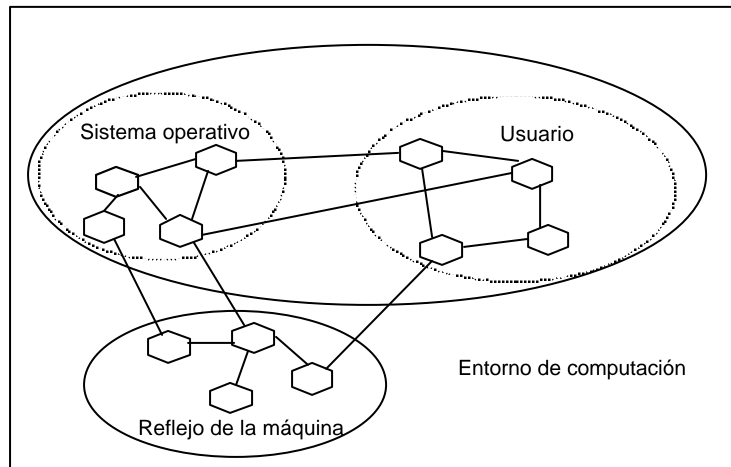


Figura 6.2. Espacio único de objetos homogéneos.

6.3.1.5 Identificador de objetos único, global e independiente

Los objetos dispondrán de un identificador único dentro del sistema. Este identificador será válido dentro de todo el sistema y será el único medio por el que se pueda acceder a un objeto. Además, el identificador será independiente de la localización del objeto, para hacer transparente su localización al resto del sistema.

6.4 Resumen

El sistema integral orientado a objetos tiene que se construido de tal forma que las características que no se consideren esenciales puedan introducirse de una manera sencilla. Tal es el caso de la distribución de los objetos. Su introducción en los niveles más bajos del sistema integral hace que sea costoso y difícil adaptar la distribución a entornos concretos, con el problema añadido de un núcleo mucho más complejo y difícil de mantener.

La solución pasa por dejar en el núcleo la funcionalidad mínima, a la manera de un micronúcleo, y colocando la funcionalidad que la extiende al nivel del sistema operativo. Dado que la nueva funcionalidad puede necesitar modificar el comportamiento del núcleo, éste se construye con una arquitectura reflectiva, que expone el propio núcleo como un conjunto de objetos.

El sistema integral queda finalmente definido por una máquina abstracta orientada a objetos, que proporciona el soporte básico del modelo único de objetos, y un sistema operativo orientado a objetos, que extiende su funcionalidad de manera transparente.

CAPÍTULO 7 ARQUITECTURA DE REFERENCIA DE LA MÁQUINA ABSTRACTA

7.1 Introducción

A continuación se describe una arquitectura de referencia de la máquina abstracta orientada a objetos con las propiedades necesarias para dar soporte a un sistema integral orientado a objetos.

7.2 Propiedades fundamentales de una máquina abstracta para un SIOO

Las características fundamentales de la misma se derivan de las características fundamentales de un SIOO vistas en el capítulo 5 y el objetivo principal de incluir la distribución de objetos como una de dichas características.

Modelo único de objetos implementado por la máquina

La máquina abstracta que se utilice deberá proporcionar las siguientes características del modelo único de objetos:

- Identidad única de objetos
- Clases y relaciones de herencia múltiple (es-un), agregación (es-parte-de) y relaciones generales de asociación (asociado-con).
- Comprobación de tipos, fundamentalmente en tiempo de ejecución
- Polimorfismo o paso de mensajes, con enlace dinámico.
- Excepciones

Uniformidad en la OO, único nivel de abstracción

La integración del concepto de proceso en la abstracción de objeto, promueve la definición de una entidad activa, el objeto, y elimina la necesidad de varios niveles de abstracción (objetos, procesos), facilitando la comprensión y utilización del modelo.

Interfaz de alto nivel, independiente de estructuras de bajo nivel

El lenguaje máquina de la arquitectura se basa en la utilización exclusiva de objetos. Es un lenguaje orientado a objetos puro de bajo nivel que será el lenguaje intermedio de la máquina abstracta. Permite declarar clases, definir métodos y manejar excepciones.

Junto con este conjunto de instrucciones de alto nivel, la máquina abstracta propuesta presenta una serie de clases primitivas definidas de forma permanente en el área de clases. Estas clases primitivas forman también parte de la interfaz de alto nivel que la máquina ofrece a las capas superiores ya que algunas operaciones básicas que en las máquinas tradicionales se ofrecen como instrucciones del lenguaje ensamblador, se ofrecen en la máquina propuesta como métodos de clases primitivas, lo que permite ofrecer un conjunto de instrucciones muy reducido.

Juego de instrucciones reducido

Según esto, el conjunto de instrucciones debe ser el mínimo posible, intentando en todo caso que, la introducción de nuevas características no provoque la introducción de nuevas instrucciones.

Flexibilidad (extensión)

La implantación de la máquina debe hacerse de tal forma que permita actuar sobre sí misma, lo que confiere flexibilidad al entorno integral. Esto se logra, fundamentalmente, haciendo que parte de la funcionalidad de la máquina se ofrezca como objetos normales con los que trabajar mediante referencias y gracias a la definición de un mecanismo de paso de mensajes que permita el paso de control a objetos que implementan parte de la funcionalidad de la máquina.

7.3 Estructura de referencia

A continuación se describen brevemente los elementos básicos en que se puede dividir conceptualmente la arquitectura. El objetivo de esta estructura es facilitar la comprensión del funcionamiento de la máquina, no necesariamente indica que estos elementos existan como tales entidades internamente en una implantación de la máquina.

Los elementos básicos que debe gestionar la arquitectura son:

- **Clases**, que se pueden ver como agrupadas en un **área de clases**. Las clases contienen toda la información descriptiva acerca de las mismas.
- **Instancias**, agrupadas en un **área de instancias**. Donde se encuentran las instancias (objetos) de las clases definidas en el sistema. Cada objeto será instancia de una clase determinada.
- **Referencias**, en un **área de referencias**. Se utilizan para realizar la invocación de métodos sobre los objetos y acceder a los objetos. Son la única manera de acceder a un objeto (no se utilizan direcciones físicas). Las referencias contienen el identificador del objeto al que apuntan (al que hacen referencia). Para la comprobación de tipos, cada referencia será de un tipo determinado (de una clase).
- **Referencias del sistema**. Un conjunto de referencias especiales que pueden ser usadas por la máquina. Se trata de referencias a objetos que realizan alguna tarea especial en el sistema.
- **Jerarquía de clases básicas**. Existirán una serie de clases básicas en el sistema que siempre estarán disponibles. Serán las clases básicas del modelo único de objetos del sistema.

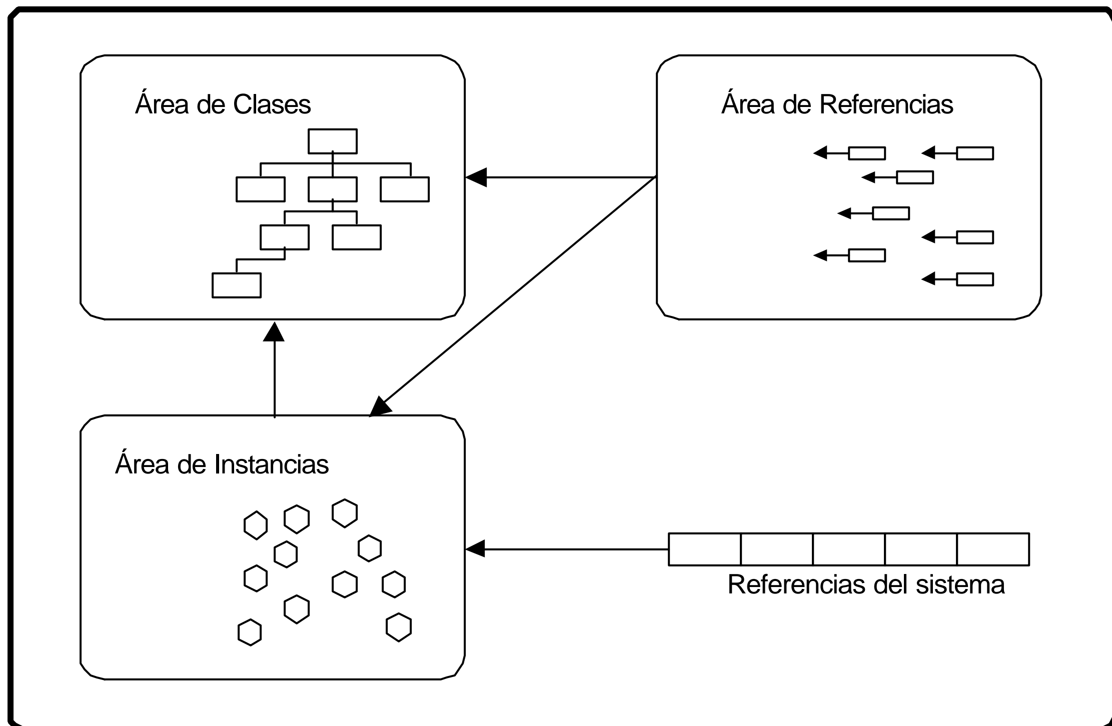


Figura 7.1. Estructura de referencia de una máquina abstracta para el sistema integral.

7.4 Juego de instrucciones

El juego de instrucciones OO de alto nivel deberá ser independiente de estructuras internas de implementación. Para ello se describirá en términos de un lenguaje ensamblador.

7.4.1 Instrucciones declarativas

La unidad de descripción en este Sistema Integral Orientado a Objeto son las **clases** y por ello, la arquitectura de la máquina abstracta almacena la descripción de las mismas en el **área de clases**.

Para la definición de clases, el lenguaje de la máquina abstracta ofrece un conjunto de instrucciones declarativas, cuyo resultado es la descripción de la información de una clase.

Estas instrucciones sirven para definir completamente y sin ambigüedad una clase, lo que supone identificar la clase de forma unívoca mediante un **nombre** que no esté asociado ya a alguna clase, especificar las **relaciones** entre distintas clases, tanto de **herencia**, como de **agregación** o de **asociación**, y, por último, definir la **interfaz** de la clase, o conjunto de métodos que la clase ofrece al exterior.

7.4.2 Instrucciones de comportamiento

Permiten definir el comportamiento de la clase, es decir, el comportamiento de sus métodos. La descripción del código que compone el cuerpo de un método supone la enumeración ordenada de un conjunto de instrucciones de entre las que se indican a continuación.

- Instrucciones para manipulación de objetos a través de las referencias

- Invocación de método a través de una referencia. Son, en esencia, la única operación que se puede realizar en un objeto y suponen la ejecución del método invocado. La ejecución de los métodos se representa mediante un objeto interno de la máquina denominado **hilo** que, virtualmente, se encuentra almacenado en el **área de hilos**.
- Control de flujo. Instrucciones para controlar el flujo de ejecución de un método entre las que se encuentran la **finalización de un método**, los **saltos** y las **instrucciones de control de flujo para gestionar las excepciones**.

En el Anexo A se muestran las distintas instrucciones del ensamblador soportadas por la implantación actual de la máquina.

7.5 Soporte básico al modelo de objetos

La máquina abstracta proporciona el soporte básico a la existencia, la identificación y el comportamiento de los objetos.

7.5.1 Modelo básico de objetos

Dentro de la máquina abstracta propuesta, el único elemento existente en tiempo de ejecución es el **objeto** y la máquina define un objeto como una **instancia de una clase en tiempo de ejecución**. Por tanto, la consecución de características como la herencia múltiple o la agregación, dependerá directamente de la forma en que la máquina abstracta represente y gestione las clases.

Además, todo trabajo con objetos se hace siempre a través de una **referencia** al mismo, nunca sobre la instancia propiamente dicha.

7.5.1.1 Heredero de las metodologías OO

Si se tiene en cuenta que la principal labor de una arquitectura orientada a objetos es la de dar soporte a los lenguajes orientados a objetos, pero que a su vez la labor de éstos es la de dar soporte a las metodologías de análisis y diseño orientado a objetos, la forma de declarar las clases en la máquina abstracta, podría tomarse de dichas metodologías (OOA, Métrica versión 3 [CSI2000], Método Unificado (*Unified Method*) [JBR99], OMT [Rum96], Booch [Boo94], etc…).

Aunque con diferente notación, todas las metodologías anteriores coinciden en la representación de los objetos (definiendo en ellos sus variables miembro y sus métodos) y las relaciones que hay entre ellos.

Estas relaciones básicamente se pueden agrupar en tres tipos, y, siguiendo la terminología de OOA, son:

- **Generalización/Especialización** o **Extensión o Derivación o Herencia** (o *is a*).
- **Todo/Parte** (o *aggregation*). Se refiere al hecho de que, en la composición de una clase, intervengan otras cualesquiera. En el caso de la arquitectura propuesta, en la que no existen tipos básicos, ésta relación incluye también a los atributos de la clase.
- **Asociación** (o *association*). Brevemente, se puede describir como aquella relación entre dos clases que no se puede ubicar en ninguna de las dos anteriores.

7.5.1.2 Declaración de clases en la arquitectura propuesta

Para cada clase que se quiera definir, será necesario enumerar ciertas características de las clases que la definan completamente y sin ambigüedad: el nombre, las relaciones con otras clases entre las que se incluyen las jerárquicas, de agregación y de asociación, así como el conjunto de métodos.

- **Nombre de la clase**. En primer lugar, es necesario dar un nombre a la clase. Este nombre deberá ser único y se almacenará, junto con el resto de la información que le siga, en el área de clases.
- **Relaciones con otras clases**. En segundo lugar, se especifican aquellas clases con las que la clase en cuestión guarda alguna relación.
  - **Herencia múltiple.** Se enumeran todas las clases de las que hereda la clase que se está definiendo, encargándose la máquina abstracta de solucionar los conflictos de variables miembro y métodos con el mismo nombre. Como regla general, se accede siempre a la variable o método de la clase que primero aparezca nombrada. En caso de que se desee acceder al otro método, se debe especificar, explícitamente, la clase a la que pertenece antes del nombre del método.
  - Toda **derivación** es **virtual**, lo que supone que, si aparece la típica estructura de herencias como la de la Figura 15.2, la clase D solo tiene una instancia de la clase A, de manera que se mantiene sincronizada respecto a los cambios que se puedan hacer desde B y desde C.

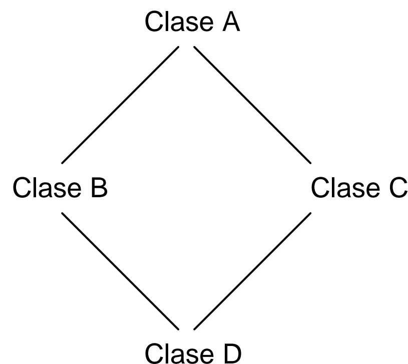


Figura 7.2. Derivación Virtual.

  - **Agregación**. Se enumeran los objetos que pertenecen a la clase junto con el nombre que se les da a cada uno. Los objetos agregados se crean cuando se crea el objeto que los contiene y se destruyen cuando éste se destruye. Son equivalentes a declarar en C++ un objeto como variable miembro de otro.
  - **Asociación.** Se declaran también los objetos que participan de una relación de asociación con la clase que se está definiendo. A diferencia de los anteriores, estos objetos no se crean automáticamente al crear el objeto ni tampoco se destruyen automáticamente al borrarlo, sino que todo ello es responsabilidad del programador. El equivalente en C++ sería declarar un puntero al tipo deseado, que es responsabilidad del programador gestionar (asignarle un objeto, apuntar a otro objeto distinto si la relación se traspasa a otro individuo, destruirlo si es que es nuestra labor hacerlo, etc.).

- **Conjunto de métodos**. Se definen a continuación los métodos que componen la clase. La declaración de estos especificará su nombre, parámetros identificados mediante la clase a la que pertenecen y valor de retorno si es que existe.

7.5.2 Polimorfismo

El polimorfismo o paso de mensajes, se logra gracias a que, la única posibilidad que ofrece la máquina para ejecutar métodos, es la invocación de los mismos.

Si a ello se suma que es en tiempo de ejecución cuando se resuelve el objeto, método y parámetros de la invocación, da como resultado que el resultado de una línea de código sólo puede ser conocido en tiempo de ejecución.

7.5.3 Excepciones

Actualmente, las excepciones se consideran ya un elemento más de cualquier sistema de computación por lo que se ha optado por incluir en la propia máquina el tratamiento de excepciones.

7.6 Ventajas del uso de una máquina abstracta

Con la utilización de una máquina abstracta que siga la arquitectura de referencia anterior se consiguen una serie de ventajas:

7.6.1 Portabilidad y heterogeneidad

La utilización de una máquina abstracta dota de portabilidad al sistema. El juego de instrucciones de alto nivel puede ejecutarse en cualquier plataforma donde este disponible la máquina abstracta. Por tanto, los programas escritos para máquina abstracta son portables sin modificación a cualquier plataforma. Basta con desarrollar una versión (emulador o simulador) de la máquina para ejecutar cualquier código de la misma sin modificación: el código es portable y entre plataformas heterogéneas. Como todo el resto del sistema estará escrito para esta máquina abstracta, el sistema integral completo es portable sin necesidad de recompilar ni adaptar nada (únicamente la máquina abstracta).

7.6.2 Facilidad de comprensión

La economía de conceptos, con un juego de instrucciones reducido basado únicamente en la OO facilita la comprensión del sistema. Está al alcance de muchos usuarios comprender no sólo los elementos en el desarrollo, si no también la arquitectura de la máquina subyacente. Esto se facilita aún más al usarse los mismos conceptos de la OO que en la metodología de desarrollo OO.

7.6.3 Facilidad de desarrollo

Al disponer de un nivel de abstracción elevado y un juego de instrucciones reducido, es muy sencillo desarrollar todos los elementos de un sistema integral orientado a objetos, por ejemplo:

7.6.4 Compiladores de lenguajes

Es muy sencillo desarrollar compiladores de nuevos lenguajes OO o lenguajes ya existentes. La diferencia semántica que debe salvar un compilador entre los conceptos del lenguaje y los de la máquina es muy reducida, pues el alto nivel de abstracción OO

de la máquina ya está muy cercano al de los lenguajes. El esfuerzo para desarrollar compiladores se reduce.

7.6.5 Implementación de la máquina

En lo que concierne a la implementación de la propia máquina, también se obtienen una serie de ventajas.

Esfuerzo de desarrollo reducido

El juego de instrucciones reducido hace que el desarrollo de un simulador de la máquina abstracta sea muy sencillo. No hay que programar código para un número muy grande de instrucciones, con lo que el tiempo necesario y la probabilidad de errores disminuyen.

Rapidez de desarrollo

Además, al ser la interfaz de la máquina independiente de estructuras internas, se puede elegir la manera interna de implementarla más conveniente. Para desarrollar rápidamente una implementación se pueden utilizar estructuras internas más sencillas aunque menos eficientes.

Facilidad de experimentación

Todo ello facilita la experimentación. Se puede desarrollar rápidamente una máquina para una nueva plataforma para hacer funcionar el sistema. Posteriormente, debido a la independencia de la interfaz, se puede experimentar con mejoras internas a la máquina: optimizaciones, nuevas estructuras, etc. sin necesidad de modificaciones en las aplicaciones.

7.6.6 Buena plataforma de investigación

Todas las ventajas anteriores la constituyen en la base para una buena plataforma de investigación. La facilidad de comprensión y desarrollo, y la portabilidad y heterogeneidad permitirán que más personas puedan acceder al sistema sobre cualquier equipo y utilizarlo como base para la investigación en diferentes áreas de las tecnologías OO: el ejemplo anterior de lenguajes OO, bases de datos, etc. Esto se aplica también a la propia máquina en sí, cuya estructura permite una fácil experimentación con diferentes implementaciones de la misma.

7.7 Minimización del problema del rendimiento de las máquinas abstractas

Como inconveniente de la utilización de máquinas abstractas se cita comúnmente el escaso rendimiento de las mismas [SS96]. Se manejan cifras que otorgan a los intérpretes una velocidad entre uno y dos órdenes de magnitud más lenta que el código compilado [May87]. En muchos casos las diferencias muy exageradas son en casos extremos o en comparaciones viciadas de programas OO que por su estructura no se pueden comparar con otros en lenguajes convencionales como C.

En cualquier caso, la propia naturaleza de una máquina abstracta necesita la utilización de un programa para simularla, con lo que el rendimiento tiene que ser menor que si se usase el hardware directamente.

Sin embargo, existen una serie de razones que hacen que este problema del rendimiento no sea tan grave, e incluso llegue a no tener importancia:

7.7.1 Compromiso entre velocidad y conveniencia aceptado por los usuarios

El simple rendimiento no es el único parámetro que debe ser tenido en cuenta en un sistema. Lo verdaderamente importante es la percepción que tengan los usuarios de la utilidad del sistema, que es función del esfuerzo de programación, la funcionalidad de las aplicaciones, y el rendimiento conseguido. El éxito que ha alcanzado la plataforma Java [KJS96] está basado en la utilización de una máquina abstracta. Esto demuestra que el compromiso entre el rendimiento y la conveniencia de los beneficios derivados del uso de una máquina abstracta ya es aceptado por los usuarios con implementaciones sencillas de una máquina abstracta.

7.7.2 Mejoras en el rendimiento

Existen una serie de áreas con las que se puede mejorar el rendimiento de las máquinas abstractas, reduciendo aún más el inconveniente de su (aparente) pobre rendimiento.

- Mejoras en el hardware
- Optimizaciones en la implementación de las máquinas. Compilación dinámica (justo a tiempo)
- Implementación en hardware

7.8 Resumen

Las propiedades fundamentales que debe tener una máquina abstracta que de soporte a un SIOO son el modelo único de objetos que implementará, con identificador único de objetos, la uniformidad en la OO, una interfaz de alto nivel con un juego de instrucciones reducido y la flexibilidad. Una estructura de referencia para este tipo de máquina abstracta se compone de cuatro elementos fundamentales: áreas para las clases, instancias, referencias para los objetos; referencias del sistema y jerarquía de clases básicas. El juego de instrucciones permitirá describir las clases (herencia, agregación, asociación y métodos) y el comportamiento de los métodos, con instrucciones de control de flujo y excepciones, creación y borrado de objetos e invocación de métodos.

Las ventajas del uso de una máquina abstracta como esta son básicamente la portabilidad y la heterogeneidad, y la facilidad de comprensión y desarrollo, que la hacen muy adecuada como plataforma de investigación en las tecnologías OO. El inconveniente de la pérdida de rendimiento por el uso de una máquina abstracta se ve contrarrestado por la disposición de los usuarios a aceptar esa pérdida de rendimiento a cambio de los beneficios que ofrece una máquina abstracta. Por otro lado, mejoras en el hardware y optimizaciones en la implementación de las máquinas minimizan aún más este problema.

CAPÍTULO 8 REFLECTIVIDAD PARA LA DISTRIBUCIÓN DE OBJETOS

8.1 Introducción

Dado que la reflectividad se ha manifestado como la arquitectura más adecuada para estructurar el Sistema Integral Orientado a Objetos (SIOO), concretamente, para organizar el modo en que Máquina Abstracta Orientada a Objetos (MAOO) y Sistema Operativo Distribuido Orientado a Objetos (SODOO) cooperan, se realiza aquí un estudio pormenorizado de la misma.

En los capítulos que van desde el 9 hasta el 14 se aplican los conceptos que aquí se exponen al caso concreto de la máquina abstracta y el sistema operativo distribuido propuestos en este trabajo.

8.2 Reflexión: modificación de la estructura o el comportamiento de un sistema

En términos humanos, **reflexión** (del inglés reflection) se refiere normalmente al acto de pensar acerca de las propias ideas, acciones y experiencias.

En el campo de los sistemas de computación, la reflectividad apareció en primer lugar en el campo de la Inteligencia Artificial aunque se propagó rápidamente a otros campos como los lenguajes de programación, donde destaca el trabajo [Smi82], y las Tecnologías Orientadas a Objeto (TOO), donde fue introducido por Pattie Maes en [Mae87].

Existen varias definiciones de **reflectividad** aunque quizá la más extendida, con algunas modificaciones, encaje con la siguiente [BGW93]:

"Reflectividad es la capacidad de un programa de manipular, como si de datos se tratase, la representación del propio estado del programa durante su ejecución".

En esta manipulación existen dos aspectos fundamentales: **introspección** e **intervención**. La introspección es la capacidad de un programa de observar y razonar acerca de su propio estado, sin posibilidad de manipularlo. La intervención es la capacidad de un programa de modificar su propio estado de ejecución o alterar su propia interpretación [BGW93, MJD96].

Ambos aspectos requieren un mecanismo que codifique el estado de ejecución como datos. La **exposición**, del inglés *reification*, es proporcionar tal codificación.

En un sistema de computación, la reflectividad es, por tanto, "*el proceso de razonar acerca de y actuar sobre el propio sistema*" [Mae88]. Esto implica no sólo poder conocer la estructura interna de parte o todo el sistema, sino además, ser capaz de modificar alguna o todas sus partes.

Formalmente, la reflectividad fue introducida en la computación por Brian Smith [Smi82] que la definió como "*la capacidad de una entidad de representar, operar o*

tratar de cualquier modo consigo misma de la misma forma que representa, opera y/o trata con entidades del dominio del problema". Dicho de otra manera, la reflexión es la capacidad de un sistema de manipular, de la misma forma que manipula los elementos que representan el problema del mundo real, elementos que representan el estado del sistema durante su propia ejecución.

8.3 Sistema base y meta-sistema

Conceptualmente, un sistema de computación puede considerarse compuesto por dos niveles: el nivel base o sistema base y el meta-nivel o meta-sistema.

El **nivel base** o **sistema base** soluciona el problema externo describiendo la computación desde el punto de vista de la aplicación, es decir, la computación que el sistema lleva a cabo como entidades que reflejan el mundo real.

El **meta-nivel** o **meta-sistema** mantiene información y describe cómo se lleva a cabo la computación en el nivel previo. Tiene como tarea hacer funcionar al sistema base y retornar información acerca del propio sistema y su computación [Mae88].

Puede considerarse que el nivel base tiene una determinada definición que es interpretada por el meta-nivel, que puede verse como un intérprete para el nivel base. El ejemplo más claro es el de un sistema definido mediante un programa en un lenguaje de programación que es interpretado por un meta-sistema: el intérprete del lenguaje de programación.

Desde el punto de vista de los lenguajes de programación, puede considerarse el nivel base como un programa escrito en un lenguaje de programación y el meta-nivel como el intérprete del lenguaje que da vida a ese programa. Este ejemplo también se adapta a una máquina abstracta orientada a objetos (o a un sistema OO en general). La máquina es el meta-sistema que da soporte a los objetos que forman el sistema base. O visto de otra manera, la máquina es un intérprete de los programas escritos en el lenguaje de la máquina (juego de instrucciones).

Las entidades que trabajan en el nivel base se denominan **entidades base**, mientras que las entidades que trabajan en otros niveles (meta-niveles) se llaman **meta-entidades** [Caz98, YW89, Fer89, ZC96].

8.3.1 Torre reflectiva

Es, pues, común pensar en un sistema reflectivo estructurado o compuesto, desde un punto de vista lógico, por dos o más niveles, que constituyen una **torre reflectiva** [WF88]. Cada uno de estos niveles sirven como **nivel base** o dominio del problema para el nivel superior y como **meta-nivel** para el nivel inferior, confiriéndole ciertas propiedades.

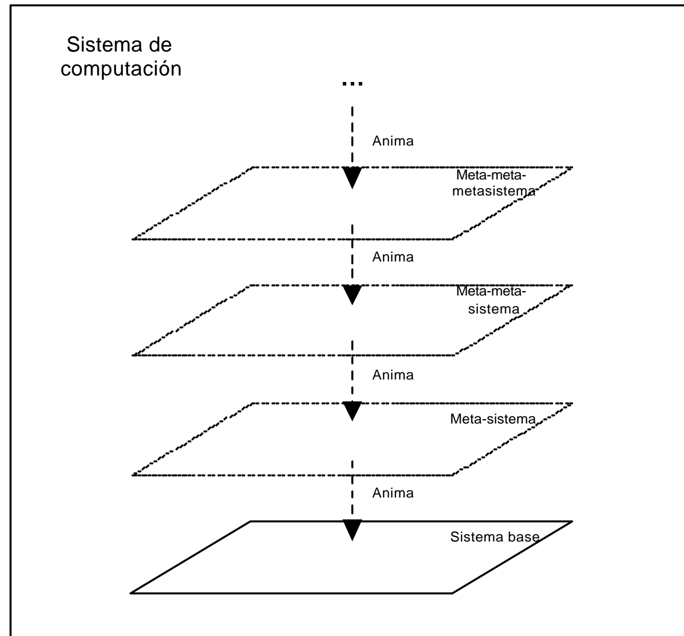


Figura 8.1. Torre reflectiva de meta-sistemas

La parte base soluciona el problema externo mientras la parte reflectiva o meta-sistema mantiene información y determina la implantación de la parte base.

8.4 Arquitecturas reflectivas

El principal motivo de añadir reflexión a los sistemas de computación es **hacer que la computación del nivel base** (el nivel o dominio del problema del mundo real que se intenta solucionar) **se lleve a cabo de forma adecuada para la resolución de un problema del mundo real**.

Para lograr crear un entorno adecuado al nivel base, es necesario hacer posible al nivel base no sólo la **consulta sino incluso la modificación de la organización interna del sistema de computación que lo soporta o meta-sistema** [YW89] facilitando el acceso a esta estructura interna como si fuesen datos, ofreciendo los mecanismos necesarios para razonar acerca de su estado actual y posibilitando la modificación del mismo para adecuarse a las condiciones actuales.

El proceso de razonar y actuar sobre el sistema implica, por tanto, el **acceso y modificación de parte o de todo el meta-sistema**.

Una arquitectura reflectiva debe proporcionar un medio para implantar la computación reflectiva de **manera modular**, lo que hace al sistema más manejable, comprensible y fácil de modifcar.

8.4.1 Reflejo del meta-sistema o meta-modelo

Uno de los problemas que se le suele achacar a la reflectividad es la posibilidad de manipulación directa y sin control de la estructura interna del sistema reflejado, lo que derivaría rápidamente en el caos.

Sin embargo, los mecanismos de reflectividad no permiten la manipulación directa del sistema de computación, sino que se ofrece un **modelo o representación**

restringida del sistema (modelo del meta-sistema[8] o **meta-modelo**) que sí está permitido cambiar y manipular [YW89, Riv88].

Este meta-modelo que se muestra al sistema base es en el que se ve reflejado el meta-sistema. Es decir, cuando el sistema base "mira" al meta-sistema que le da vida para conocer información acerca de sí mismo, ve su "**reflejo**[9]" (la representación que el meta-sistema usa para mostrarse al sistema base). De ahí el nombre de reflectividad[10] para esta propiedad.

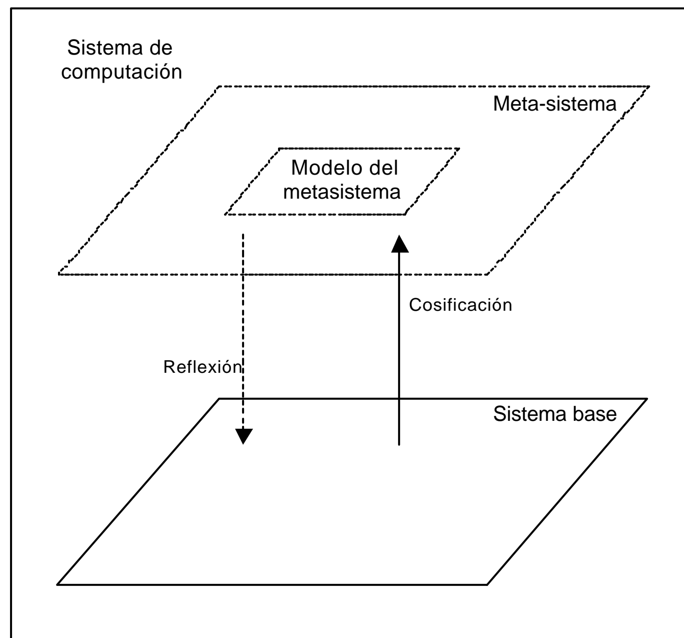


Figura 8.2. Interacción entre el sistema base y el meta-sistema mediante un modelo del meta-sistema.

8.4.1.1 Requisitos del modelo

El modelo del sistema debe satisfacer varios requisitos para exponer completamente los aspectos del sistema elegidos y permitir la modificación de los mismos de forma efectiva y segura [Fer89, YW89]: un nivel de abstracción adecuado, sobre el que sea fácil razonar y manipular, manteniendo la relación de causalidad del modelo con el sistema.

- **Abstracción adecuada del sistema**. El nivel de abstracción que el modelo aplica sobre el sistema debe ser adecuado con relación al uso que se pretenda hacer de la reflectividad. Para llevar a cabo la reflectividad en un sistema es necesario definir qué aspectos se quieren reflejar en el modelo, es decir, qué

---

[8] Aunque no se haga referencia explícita, se entiende que cualquier mención al meta-sistema cuando se cambia o manipula es siempre mediante su modelo o representación.

[9] El reflejo es el resultado de la reflexión (la acción de reflejar). En inglés se denominan con la misma palabra reflection.

[10] Otra traducción podría ser reflexividad, entendiendo esta propiedad como la capacidad de reflejar imágenes. También se justifica desde el punto de vista del acceso del sistema a su propia representación, de manera reflexiva.

entidades y/o características del meta-sistema deben exponerse. Estas entidades o aspectos del sistema de computación en sí serán aquellos sobre los que actúa la parte reflectiva de la arquitectura, para lo que deben representarse o transformarse en algo que pueda ser manipulado.

- **Adecuado para el razonamiento y la manipulación**. El modelo debe representar al sistema de forma que sea sencillo razonar sobre él y su estado, así como la manipulación del mismo. De esta forma, el sistema base debe poder manipular el modelo del meta-sistema (**meta-interacción**).
- **Relación de causalidad. Conexión causa-efecto entre el modelo o representación del meta-sistema y el meta-sistema en sí.** El modelo debe estar conectado con el sistema mediante una relación de causalidad, de tal forma que los cambios efectuados sobre el modelo tengan su reflejo en el comportamiento posterior del sistema.

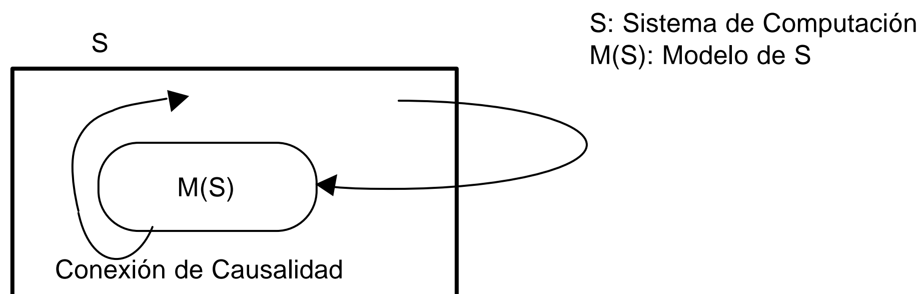


Figura 8.3. Sistema Reflectivo. El modelo y la relación de causalidad

8.4.2 Actividades básicas en un sistema reflectivo: exposición y reflexión

En el proceso de reflexión se identifican dos actividades básicas: **exposición**[11], (del inglés, *reification*) y **reflexión**, (del inglés, *reflection*) [Mae87].

8.4.2.1 Exposición

Los dos aspectos básicos de la reflectividad, **introspección** e **intervención** [BGW93, Mae87], requieren un mecanismo que codifique el estado de ejecución de ciertos aspectos del sistema como datos accesibles, creando un modelo del sistema, a la vez que ofrecen mecanismos adecuados para la manipulación de tales aspectos de sí mismo [Mae87].

La **exposición** es la capacidad de proporcionar tal representación, transformando y haciendo accesibles a los elementos del nivel base aquellos aspectos del meta-sistema que, normalmente, le son intangibles [BGW93, CS98].

Como nota adicional, supóngase un sistema orientado a objetos (LPOO, MAOO) que exponga la estructura interna de los objetos, es decir, que codifica, mediante objetos de primera clase, la estructura interna de los objetos.

[11] Además del término exposición, a lo largo de este documento también se pueden encontrar los términos cosificación y concretización. Todos ellos pueden encontrarse como traducciones del término reification en el Anexo E.

Por ejemplo, la API reflectiva de Java [Sun97b] define nuevas clases (field, method, constructor, etc.), para representar los distintos componentes que forman la representación en tiempo de ejecución de un objeto.

Los objetos instancias de estas clases representarán los métodos, campos, etc., del objeto en tiempo de ejecución.

8.4.2.2 Reflexión

La **reflexión** es el acto de modificar algunos aspectos del sistema por el sistema en sí **gracias a la actuación sobre el modelo ofrecido**. De esta forma, cualquier cambio en el modelo, en los aspectos expuestos se refleja en el comportamiento del sistema [CS98, YW89].

Más aún, si es posible modificar el objeto u objetos que describen la estructura interna de los objetos del sistema, se modificaría la representación de los mismos en tiempo de ejecución.

Por ejemplo, la API reflectiva de Java no sólo define los objetos que en tiempo de ejecución representarán las distintas parte de una clase, sino también los medios para extraer y manipular tales objetos. Para ello, añade a la clase Class los métodos que permiten extraer información de la misma, por ejemplo, getMethods o getfields, que extraen los métodos que forman la interfaz de la clase y los campos que componen su estado interno, respectivamente, retornando objetos instancia de las clases Method y Field. Los métodos que componen la interfaz de esas clases permiten consultar y modificar sus valores. Por ejemplo, los métodos de la clase Field permiten obtener el tipo del campo subyacente, interrogar al objeto acerca del valor del campo en el objeto e, incluso, modificarlo.

Existe pues una correspondencia entre los aspectos reflejados y su representación, de tal forma que cualquier modificación en la representación modifica el aspecto representado y viceversa [YW89].

8.5 Aplicación de la reflectividad en sistemas orientados a objetos

La reflectividad encaja especialmente bien con los principios promovidos por las tecnologías Orientadas a Objetos [BG96] que insisten en la encapsulación en los niveles y la modularidad en los efectos.

En el caso de un sistema orientado a objetos, el paradigma mediante el que se estructura el sistema es la orientación a objetos. Para introducir la reflectividad en el sistema de manera uniforme, lo más interesante es **que el modelo del meta-sistema esté descrito utilizando el mismo paradigma OO que el propio sistema base**. De este modo, tanto el modelo, como las operaciones de exposición y reflexión pueden ser utilizados de manera uniforme dentro del mismo paradigma.

En el nivel base se encontrarán los **objetos normales** del sistema y en el meta-nivel se encuentran los objetos que dan soporte a los objetos base u **objetos del meta-nivel**.

En un sistema OO, un **objeto base** u **objeto del nivel base** es un objeto definido por la aplicación y que se refiere al ámbito del problema, mientras que un **objeto del meta-nivel** es un objeto que da soporte a los objetos del nivel base para lo que representan información acerca de los mismos y determinan su comportamiento [Fer89].

Un conjunto de objetos del meta-nivel que dan soporte a un objeto base determinado se denomina **meta-espacio**.

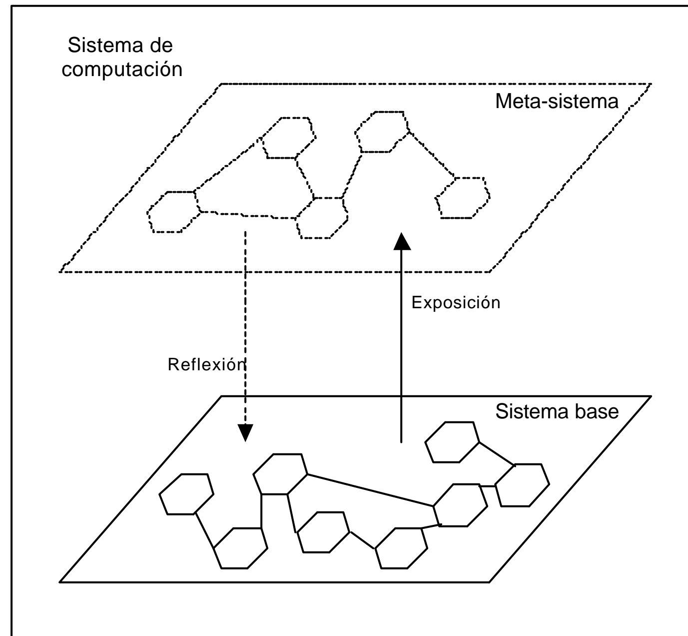


Figura 8.4. Descripción del meta-sistema mediante un modelo de objetos.

8.5.1 Representación del modelo del sistema como un conjunto de objetos

Extendiendo las ideas anteriores, la construcción de un modelo que permita la interacción entre el nivel base y el meta-nivel debe hacerse también siguiendo el principio de la OO.

A los objetos que componen el modelo del meta-sistema se le denomina **meta-objetos**, cada uno de los cuales expone ciertos aspectos de la estructura del meta-sistema subyacente [Foo90].

La bondad de esta idea es fundamental para la adopción de una arquitectura reflectiva como mecanismo de estructuración del sistema operativo y la máquina abstracta OO, punto fundamental en este trabajo.

Es posible organizar el control del comportamiento de un sistema de computación OO o meta-sistema OO, su **meta-interfaz**, a través de un conjunto de objetos que pueden representar varias características del contexto de ejecución tales como representación, ejecución, comunicación o localización, entre otras [BG96, WY88, Fer89].

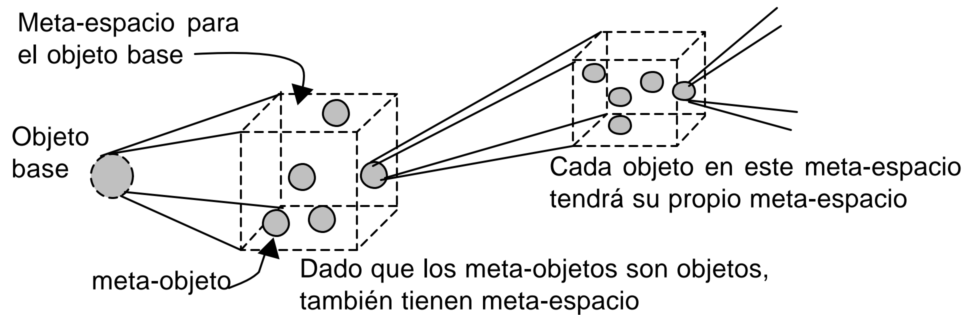


Figura 8.5. Relación objeto/meta-objeto.

Especializando estos objetos, o **meta-objetos**, es posible extender y modificar el contexto específico de ejecución de objetos de la aplicación. Así, un meta-objeto no sólo mantiene información acerca de los objetos del nivel base sino que además controla su ejecución.

De esta forma se adapta el entorno de ejecución (el meta-sistema) a los requerimientos particulares de cada objeto o conjunto de objetos de una aplicación, eliminando los dilemas y conflictos de correspondencia que se producen cuando la implantación de una característica no se ajusta a los requerimientos necesarios para la adecuada ejecución de la aplicación.

La reflectividad también puede ayudar a expresar y controlar la gestión de recursos, como la planificación.

8.5.1.1 Uniformidad

La representación del modelo del sistema como una colección de objetos hace que la utilización de las capacidades reflectivas por parte de una aplicación se haga por medio del paso de mensajes a los objetos del modelo manteniendo la uniformidad conceptual alrededor del concepto de objeto [YW89, Fer89]

8.6 Meta-interacción: transferencia de control

Una parte fundamental de una arquitectura reflectiva es el mecanismo de transferencia de control desde el nivel-base al meta-nivel, es decir, bajo qué condiciones un sistema manipulará su auto-representación y qué clase de computación llevará a cabo el sistema cuando se den esas condiciones [Mae88].

8.6.1 Propiedades

El mecanismo de transferencia de control debe cumplir algunas propiedades como la transparencia y el sincronismo.

- **Transparencia**. La transferencia de control debe ser transparente al objeto del nivel-base, es decir, el programador no debería necesitar considerar ninguna funcionalidad del meta-nivel a la hora de diseñar e implementar el objeto del nivel base.
- **Sincronismo**. La transferencia de control debería ser síncrona, lo que da al meta nivel la oportunidad de controlar la ejecución concurrente del nivel base. Diferentes mecanismos de sincronización pueden implantarse como programas al meta-nivel.

8.6.2 Clasificación de arquitecturas reflectivas atendiendo a la transferencia de control

Se pueden identificar dos tipos de arquitecturas, dependiendo del momento y condiciones en que se transfiere el control del nivel-base al meta-nivel: **reflectividad explícita** y **reflectividad implícita** [Mae88].

8.6.2.1 Reflectividad implícita

El **sistema activa sistemática y automáticamente la computación reflectiva**. Esto significa que existen agujeros en el proceso normal de interpretación que la computación reflectiva llenará, o lo que es lo mismo, si el proceso de interpretación consiste en la ejecución de una serie de tareas t_1, t_2, ..., t_n, existen uno o más i para los que la acción t_i no está definida por la máquina subyacente. Es decir, algún t_i es, necesariamente, un trozo de computación reflectiva definida por el programa que está siendo ejecutado (o interpretado).

Por ejemplo, en una máquina abstracta existe reflectividad implícita cuando se interpretan las instrucciones del programa. Cada vez que el programa usa una instrucción, la máquina realiza la computación interna necesaria para ejecutarla (la interpreta), siendo esta la manera normal de funcionamiento por la que el sistema utiliza el meta-sistema.

En la arquitectura propuesta, esta será una de las formas en las que la meta-computación se lleve a cabo.

8.6.2.2 Reflectividad explícita

La reflectividad explícita no ocurre automáticamente sino que **tiene lugar únicamente cuando la aplicación lo solicita**. De esta forma, la computación tiene lugar, normalmente, en el nivel de objetos. Siempre y cuando el programa solicite la computación reflectiva, el sistema, temporalmente, deja este nivel y pasa a realizar alguna tarea en el nivel reflectivo.

Un ejemplo es cuando el programa, de manera explícita solicita el tamaño (parte de la representación interna) de un tipo de datos del lenguaje de programación C mediante el operador `sizeof`.

8.7 Tipos de reflectividad

La integración de la reflectividad en los sistemas orientados a objetos puede hacerse de manera uniforme. Sin embargo, existen algunos trabajos en tal dirección que han llevado a dos modelos diferentes de reflectividad [Fer89]: la **reflectividad estructural** y la **reflectividad del comportamiento** o **reflectividad computacional**, que no deben considerarse excluyentes.

El tipo de reflectividad soportado por el modelo especifica cuáles son los aspectos del sistema que se exponen y sobre los que, por tanto, se puede actuar [Fer89, Caz98].

8.7.1 Reflectividad estructural

La **reflectividad estructural** se puede definir como la *capacidad de exponer tanto el programa que está siendo ejecutado en este momento como sus datos* [Caz98]. En caso de que se tratase de una arquitectura reflectiva OO, esta exposición se llevaría a cabo utilizando los conceptos de OO, como objeto o clases.

Este modelo tiene en cuenta la extensión de la parte estática de un sistema, es decir, permite exponer y manipular los aspectos estructurales del sistema de computación, como la herencia en un sistema OO, considerándolos como datos [Coi88] y ofrece mecanismos que permitan la manipulación de dicha representación.

Un ejemplo de reflectividad estructural lo constituye el operador `sizeof`, que permite averiguar el espacio ocupado por un tipo de dato en tiempo de ejecución, o la API reflectiva de Java que permite obtener los métodos y campos que componen un objeto en tiempo de ejecución, modificar el contenido de un campo, invocar un método en tiempo de ejecución, etc.

8.7.1.1 Reflectividad estructural aplicada a la OO

Los sistemas OO parecen ser uno de los campos en donde la reflectividad puede ser más fácilmente implantada, dada la naturaleza inherente de tales sistemas a distribuir en objetos los datos y procedimientos que representan el sistema.

La reflectividad estructural en sistemas OO, introducida por P. Cointe [Coi88], se refiere al uso reflectivo de clases y metaclases para implementar objetos.

En un modelo de clases puro, cada entidad es una instancia de una clase y las clases, a su vez, son instancias de otras clases, llamadas **metaclases**. Este modelo tiene en cuenta la extensión de la parte estática de los lenguajes OO, es decir, de los aspectos estructurales de los objetos considerados como la implantación de un tipo abstracto de datos [BC87].

La reflectividad estructural está muy extendida tanto en los lenguajes funcionales, como Lisp, como en los lenguajes lógicos, como Prolog, que tienen sentencias para manipular la representación del programa. Por ejemplo, Prolog representa los programas como estructuras y es posible acceder al *functor*, los argumentos, etc. Esta reflectividad se fundamenta en el hecho de que ambos son lenguajes interpretados.

En el caso de los lenguajes orientados a objetos, la mayoría son compilados y, por tanto, no hay representación de los mismos en tiempo de ejecución. Una excepción es Smalltalk, que utiliza las clases para llevar a cabo la reflectividad estructural [BC89, FJ89].

Es adecuada cuando lo que pretendemos es dotar al sistema de mecanismos como la introspección y se aplica fundamentalmente a elementos estructurales del sistema como la herencia.

Como ejemplos conocidos y significativos de reflectividad estructural podemos citar la API Reflectiva de Java [Sun97], el sistema de información de tipos en tiempo de ejecución o RTTI de C++ [Str92] o SOM [IBM96a, IBM96b].

8.7.2 Reflectividad del comportamiento

La **reflectividad del comportamiento**, introducida por P. Maes [Mae87] y B. Smith [Smi82], se refiere al comportamiento del sistema computacional y se puede definir como *la capacidad de un lenguaje – en general cualquier sistema – para proporcionar una exposición completa de su semántica y de los datos que utiliza para ejecutar el programa actual.*

La reflectividad del comportamiento manipula el comportamiento del sistema de computación en tiempo de ejecución e implica dotar a un programa P de la capacidad y los mecanismos necesarios para observar y modificar las estructuras de datos utilizadas

para su propia ejecución [MJD96]. Es decir, la reflectividad del comportamiento supone la modificación de alguna característica del propio sistema subyacente.

Un posible ejemplo es el ORB reflectivo OpenCorba [Led99] que expone, entre otras características internas del *bus*, la invocación remota. OpenCorba permite la modificación del mecanismo de paso de mensajes para implantar distintas políticas de envío lo que, lógicamente afectará a los futuros envíos de mensajes a objetos. Cuando se envía un mensaje a un objeto representante o *proxy* para su reenvío al objeto remoto al que representa, el mensaje se intercepta a su llegada al objeto representante y se reenvía al servidor remoto. Sin embargo, el control del envío al objeto remoto se realiza en el meta-nivel, donde se pueden implantar políticas que modelen Java RMI [Sun98], futuras versiones de CORBA DII o invocaciones locales.

8.7.2.1 Superposición reflectiva

La flexibilidad permitida por la reflectividad puede llegar a ser excesiva, lo que hace necesario un control sobre los cambios que permite la reflectividad en el sistema. En concreto, las modificaciones permitidas por la reflectividad pueden llevar a inconsistencias si las actualizaciones hechas por el código reflectivo, el que modifica el entorno en tiempo de ejecución, involucran aspectos del sistema que se consideran implícitos. Por ejemplo, si el sistema implanta herencia, ésta debe tener siempre el mismo significado.

Este fenómeno, conocido como **superposición reflectiva**, del inglés *introspective overlap*, es estudiado por B.C.Smith [Smi82, Smi84], que establece una distinción entre el intérprete (o sistema en tiempo de ejecución) P_1, que ejecuta el programa P, y el intérprete (o sistema en tiempo de ejecución) P_2 que ejecuta el código reflectivo.

8.7.2.2 Torre reflectiva

De hecho, este argumento puede llevarse al extremo, permitiendo código introspectivo en el intérprete P_2, que necesitará otro intérprete P_3 y así sucesivamente, posibilitando un número potencialmente infinito de intérpretes.

Esta pila de intérpretes, denominada **torre reflectiva**, no necesita estar basada en técnicas interpretativas, por lo que Smith y des Rivières [dRS84] propusieron el término alternativo de **procesador reflectivo**.

8.7.2.3 Reflectividad del comportamiento en sistemas OO

La reflectividad del comportamiento en sistemas orientados a objeto, se basa en el hecho de que cada objeto del nivel del problema tiene asociado su propio objeto que representa la información implícita acerca de su existencia: su estructura y el mecanismo de gestión de mensajes [Mae87, Smi82]. En el siguiente apartado se extenderá este concepto.

8.8 Modelos de reflectividad

Existen diferentes modelos de reflectividad que se pueden aplicar a la reflectividad estructural y del comportamiento[12], aunque algunos modelos no son demasiado apropiados para algunos tipos de reflectividad. J. Ferber en [Fer89] establece una

[12] En lo que sigue, nos referiremos siempre a la reflectividad del comportamiento a menos que, explícitamente, se diga lo contrario.

primera clasificación de los modelos de reflectividad: el modelo meta-clase, el modelo meta-objeto y la exposición de la comunicación.

8.8.1 Modelo meta-clase

Según la definición de [Gra89], *una clase describe tanto la estructura como el comportamiento de sus instancias. La estructura es el conjunto de variables cuyo valor estará asociado a las instancias. El comportamiento es el conjunto de métodos a los que las instancias de la clase serán capaces de responder*.

El modelo de **meta-clase** [Coi87, BC89] se basa en la equivalencia entre la clase de un objeto y su meta-objeto y, por tanto, implementa la torre reflectiva siguiendo la cadena de instanciaciones.

Las clases serán objetos, instancias de una clase, su **meta-clase**. A su vez, las meta-clases forman una jerarquía que deriva de la clase raíz, **CLASS**.

De esto se deriva que una clase es un meta-objeto que expone una entidad base. De hecho, en los sistemas basados en el paradigma de clase/instancia, la clase de un objeto es considerada normalmente como su meta-objeto, debido a la reflectividad estructural del modelo.

De esta forma, para un objeto O es equivalente, desde el punto de vista del comportamiento, recibir un mensaje M o que su clase reciba el mensaje indicado para gestionar un mensaje, que, en la mayoría de las obras se representa como **HANDLEMSG**.

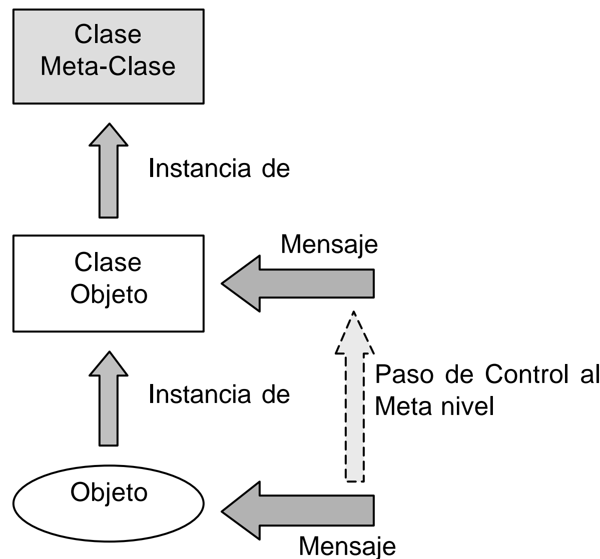


Figura 8.6. Esquema del modelo Meta-Clase.

El método HANDLEMSG se define en la meta-clase y el método de gestión de mensajes por omisión se describe en la meta-clase raíz de la jerarquía de metaclases (CLASS), de la que son instancias las clases.

Cuando un objeto recibe un mensaje, debe decidir si utilizar el mecanismo por omisión o pasar el control al meta-nivel. Esto puede lograrse mediante una entrada definida en el receptor, lo que permite lograr la reflectividad para un objeto, o en su clase, lo que establece un modo reflectivo para todas las instancias.

8.8.1.1 Ventajas

Sencillez

La clase encaja perfectamente con el papel de controlador y modificador de la información estructural porque mantiene esa información. La implantación de este modelo es posible, únicamente en aquellos lenguajes que manejen las clases como objetos, como por ejemplo Smalltalk [FJ89] o CLOS [BGW93].

8.8.1.2 Inconvenientes

Se han señalado algunos inconvenientes a este modelo [Fer89]: la dificultad de especializar el comportamiento de una instancia concreta, de modificar dinámicamente la meta-clase de un objeto por otra o de registrar información acerca del objeto en su meta representación. Además, la interacción de las meta-clases y la herencia da lugar a la aparición de problemas de compatibilidad entre distintas meta-clases.

Especialización

El problema radica en la especialización del meta-comportamiento de una instancia concreta. Todas las instancias de una clase tienen el mismo meta-objeto, por tanto, todas tienen el mismo meta-comportamiento.

Para especializar el meta-comportamiento para una instancia, es posible usar la herencia construyendo una clase nueva que difiera de la clase original sólo en el meta-comportamiento, o bien mantener diccionarios que mantengan información y métodos de cada instancia. Sin embargo, esto añade complejidad al sistema, quitando parte de la sencillez inicial.

Algunos intentos de añadir reflectividad a la máquina virtual Java [GK98a] tienen este problema.

Cambio dinámico

Otro problema radica en la dificultad de cambiar dinámicamente el comportamiento de un objeto. Tal cambio implicaría sustituir su meta-clase por otra.

Sin embargo, no todos los lenguajes permiten el cambio dinámico de la clase de un objeto. Es más, el cambio de la clase de un objeto puede derivar en inconsistencias.

Parcialmente meta

Dado que todos los objetos de una clase comparten la misma meta-clase, se ven obligados a almacenar la misma meta-información. Esto hace imposible registrar características particulares de cada objeto en su meta-representación. Esta limitación es crucial dado que uno de los aspectos más importantes de las meta-entidades es su capacidad para almacenar información acerca de su objeto base o referente.

Compatibilidad

Los lenguajes basados en clases organizan las aplicaciones en jerarquías de clases utilizando para ello la herencia, lo que además facilita la reutilización [Weg90].

Aquellos lenguajes que utilizan el concepto de clase para la reflectividad tienen que tener en cuenta, además del enlace clase-subclase-superclase, el enlace de instancia existente entre una clase y su meta-clase.

La interacción entre la herencia y la instanciación debe tenerse muy en cuenta a la hora de diseñar librerías de clases. El problema radica en determinar si una clase puede

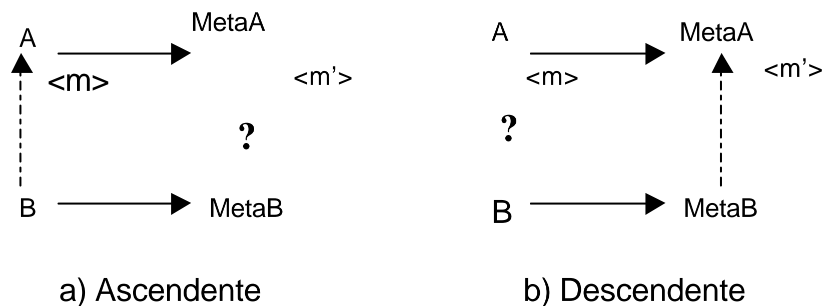
heredar de otra o no, partiendo del hecho de que ambas clases son instancias de meta-clases diferentes.

Esto da lugar a la aparición del problema de la compatibilidad de meta-clases como se apunta en [Gra89, BLR96].

- **Compatibilidad ascendente**. Considérese la situación representada en la figura 8.7(a), donde una clase `A`, instancia de una meta-clase `MetaA`, implementa un método `m` que envía el mensaje `m'` a la clase a la que pertenece la instancia que recibe el método `m`.

  Supóngase una clase `B`, derivada de la clase `A` e instancia de una metaclase `MetaB`. Por derivar de `A`, `B` tendrá entre sus métodos el método `m`. Sin embargo, si entre `MetaA` y `MetaB` no existe ninguna relación, es factible suponer que el mensaje `m'` sólo sea conocido por `MetaA`. Cabe entonces preguntarse qué sucederá cuando una instancia de `B` reciba el mensaje `m`, lo que provocará que `MetaB` reciba el mensaje `m'`, que no necesariamente está implantado como parte de su interfaz.

- **Compatibilidad descendente**. Considérese la situación representada en la Figura 8.7(b), donde una clase `A`, instancia de una meta-clase `MetaA`, implementa un método `m` y `MetaA` implementa el método `m'` que crea una nueva instancia del receptor y le envía el mensaje `m`. Supóngase que `MetaB` deriva de `MetaA`, heredando el método `m'`. Si se envía el mensaje `m'` a la clase `MetaB`, como resultado `MetaB` crearía una nueva instancia `B` y le enviaría el mensaje `m`. Si `B` no guarda ninguna relación con `A`, es factible pensar que no tenga implantado el método `m`. Cabe entonces preguntarse qué sucederá cuando `MetaB` reciba el método `m'` y, a su vez, invoque el método `m` en `B`.



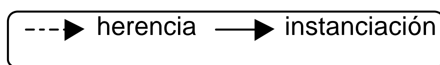


Figura 8.7. Problemas de compatibilidad del modelo Meta-Clase

8.8.2 Modelo meta-objeto

En el modelo de **MetaObjetos** [KRB91], los meta-objetos son instancias de una clase especial, la clase **MetaObjeto** o de una de sus subclases. Los meta-objetos no se identifican con la clase del objeto sino que se ligan arbitrariamente con los objetos a los que se refieren y únicamente incluyen información de control del comportamiento, dejando la información estructural a la clase. De esta forma, **la torre reflectiva se lleva a cabo siguiendo la relación cliente/servidor**.

De esta forma cuando un objeto O recibe un mensaje, es equivalente a enviar el mensaje **HANDLEMSG** a su meta-objeto.

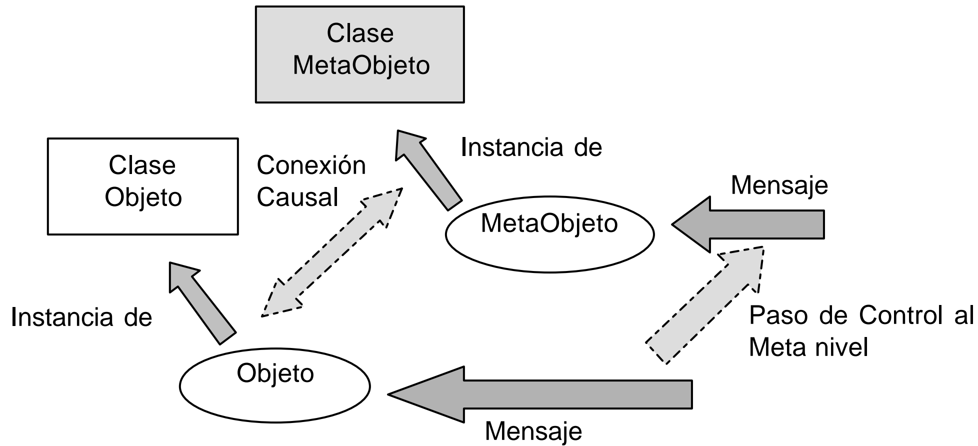


Figura 8.8. Esquema del modelo Meta-Objeto.

Entidades independientes gestionan tanto la introspección como la intervención de cada entidad base. Cada meta-objeto intercepta los mensajes enviados a su objeto y ejecuta su meta-computación antes de reenviarlos a su objeto base.

En este modelo no existen, en principio, restricciones ni imposiciones sobre la relación entre las entidades base y las meta-entidades. Cada meta-objeto puede estar conectado a varios objetos del nivel base y cada objeto puede tener enlazados varios meta-objetos durante su ciclo de vida, cada uno de los cuales se refiere a una parte concreta de su comportamiento. Al conjunto de meta-objetos que componen el entorno de ejecución de un objeto se le denomina **meta-espacio**.

8.8.2.1 Ventajas

Elimina, prácticamente, todos los inconvenientes del modelo anterior, permitiendo especializar el meta-comportamiento, monitorizar el objeto y modificar el meta-objeto.

Especialización del meta-comportamiento

Es sencillo especializar el meta-comportamiento de un objeto individual desarrollando una nueva clase de meta-objetos que refinen el comportamiento original de alguna forma.

Monitorización del objeto

También es sencillo hacer que un meta-objeto monitorice el comportamiento de su objeto, registrando qué sucede y, eventualmente, decidiendo algún cambio.

Modificación del meta-objeto

Partiendo de que los meta-objetos son objetos como cualquier otro, que se crean y destruyen de la misma forma que otros objetos, con un identificador único y conocido y sabiendo que se enlazan con su objeto base a través de alguna asociación con este identificador único, es posible modificar fácilmente el meta-objeto de un objeto asignando a tal asociación el identificador de otro meta-objeto con un comportamiento diferente.

8.8.2.2 Inconvenientes

El mayor inconveniente reseñable de este modelo es que un meta-objeto puede monitorizar un mensaje sólo cuando ya ha sido recibido por su objeto referente. Por tanto no se puede actuar sobre él durante la transmisión. Este es el modelo más usado no sólo en lenguajes de programación (3-KRS) sino también es sistemas operativos ApertOS [Yok92] y sistemas distribuidos como CodA [McA95a].

8.8.3 La reflectividad como exposición de la comunicación o exposición de los mensajes

Esta última aproximación, consiste en la exposición de la comunicación en sí. Cada comunicación puede ser un objeto y, por tanto, atiende o reacciona al mensaje **SEND**.

8.8.3.1 Los objetos mensaje

En este modelo, las meta-entidades son objetos especiales, denominados **mensajes**, que exponen las acciones que deberían ejecutar las entidades base [Fer89].

La clase **MESSAGE** contiene toda la información necesaria para interpretar el mensaje. Cuando se invoca un método, se crea una instancia de la clase **MESSAGE** y se envía el mensaje **SEND** a tal instancia. El objeto mensaje está dotado con su propia gestión de acuerdo al tipo de meta-computación requerida. Cuando la meta-computación termina, se destruye el objeto.

La clase de un mensaje define el meta-comportamiento realizado por el mensaje de forma que diferentes mensajes pueden tener diferentes clases. Es posible definir diferentes comportamientos para llamadas a métodos para cada objeto, especificando un modo diferente para cada llamada a método.

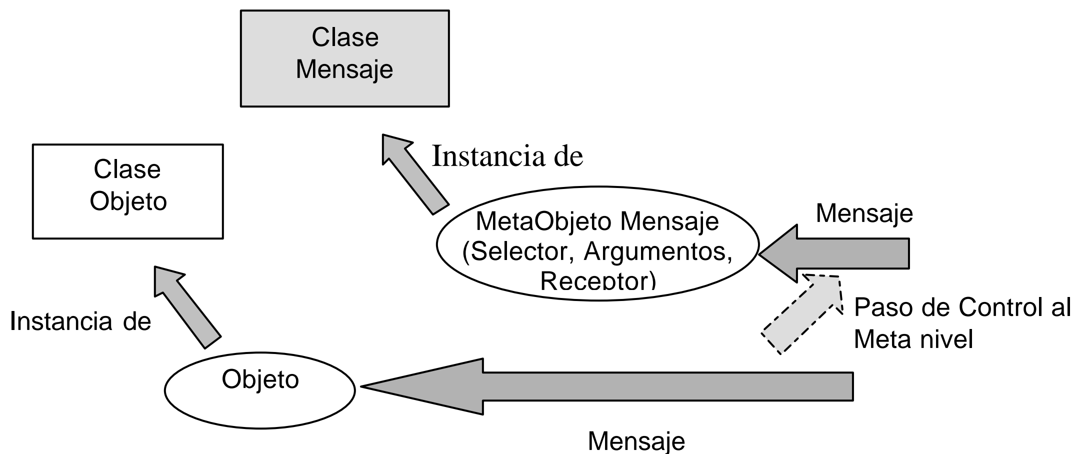


Figura 8.9. Esquema del modelo de exposición de mensajes.

8.8.3.2 La torre reflectiva

Este modelo tiene una torre reflectiva compuesta únicamente por dos niveles, el nivel base y el meta-nivel, que expone los mensajes.

8.8.3.3 Transferencia de control

En esta aproximación el sistema pasa el control al meta-nivel para todos los envíos de mensajes excepto cuando recibe el mensaje **SEND** para evitar la regresión infinita.

De este modo, el sistema está casi siempre en modo reflectivo.

8.8.3.4 Ventajas

La primera ventaja destacable es la facilidad que ofrece para diferenciar entre distintos tipos de mensajes: se pueden definir y usar distintas subclases de comunicación en lugar de la clase **MESSAGE** estándar.

Por otro lado, el uso de diferentes tipos de mensajes permite una extensión incremental del sistema. Es posible incorporar fácilmente nociones como paso concurrente de mensajes, etc.

8.8.3.5 Inconvenientes

La principal desventaja de este modelo es que los meta-objetos mensaje no están enlazados de ninguna manera con las entidades base que los originan y, por tanto, no pueden acceder a su información estructural.

Además, los meta-objetos mensaje tienen una vida muy corta, existen sólo durante el tiempo que dura la acción que representa.

Debido a todo esto, son inadecuados para monitorizar el comportamiento de los objetos del nivel base o almacenar información sobre la meta-computación (**carencia de continuidad**), para representar información específica acerca de los objetos o representar el comportamiento de un objeto determinado.

Sin embargo, es una aproximación hábil para combinarse con los modelos anteriores.

8.9 Meta-interfaces y MOP

El modelo de la **implantación abierta** propuesto por Kiczales [Kic92] argumenta que, mientras que el modelo tradicional de ocultación de información protege a los clientes de una abstracción de tener que conocer los detalles de cómo está implementada esta, también evita que pueda alterar algunos detalles cuando lo necesite.

Así pues, se manifiesta como conveniente que los clientes de una abstracción puedan tener algún control sobre decisiones tomadas en la implantación de la misma.

Sin embargo, esta posibilidad de adaptación del entorno de ejecución no debe suponer quebraderos de cabeza para los usuarios. En este punto, no está de más recordar que la razón original de intentar esconder los detalles de implantación era una muy buena razón: simplificar la tarea del programador abstrayendo aquellos detalles con los que se suponía que no tendría que trabajar [KL93].

8.9.1 Interfaz base *versus* interfaz meta

Para permitir la adaptación sin incrementar la complejidad, se propone separar la interfaz que presenta un servicio o abstracción en **interfaz base** o **protocolo base**, que ofrece a los usuarios la funcionalidad usual de la abstracción o servicio, y **meta-interfaz** o **meta-protocolo**, que permite a los clientes controlar cómo se implementan ciertos aspectos de la abstracción o servicio [KL93, GC95, HPM+97].

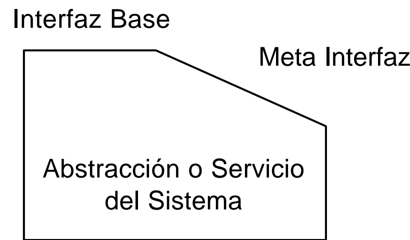


Figura 8.10. Interfaz base *versus* meta-interfaz

Por tanto, en lugar de exponer todos los detalles de la implantación, la meta-interfaz debería proporcionar una visión abstracta de la misma que permitiese a los clientes ajustarla en modos bien definidos.

El objetivo final de separar el comportamiento y la implantación es permitir al programador cliente:

- ser capaz de controlar la implantación del servicio cuando lo necesite
- ser capaz de tratar el comportamiento y la implantación por separado.

Ofreciendo interfaz base y meta-interfaz por separado, se ofrece al usuario programador una visión separada de ambos mundos.

8.9.1.1 Interfaz base

La interfaz de la aplicación se modela como una caja negra tradicional que expone la funcionalidad y no los detalles de implantación.

Por ejemplo, en el caso de la memoria virtual, la interfaz base únicamente lee y escribe bytes, en el caso de un sistema de ficheros abre/cierra ficheros, lee/escribe bytes, etc. Los clientes del servicio escriben los programas sobre la interfaz base de modo tradicional.

8.9.1.2 Meta-interfaz

La meta-interfaz articula y expone los detalles de implantación posibilitando a los clientes controlar las decisiones a tomar a la hora de establecer la correspondencia entre un servicio y su implantación final, permitiendo así conseguir una **personalización de grano fino**.

Por ejemplo, en el caso de una meta-interfaz al sistema de memoria virtual, el cliente escribiría código utilizando la interfaz base (leer y escribir bytes de o en memoria) y, en caso de que el sistema de paginación por omisión ofreciese un rendimiento pobre para el comportamiento del cliente, este podría escribir código usando la meta-interfaz para indicar un algoritmo alternativo de paginación para mejorar el comportamiento del código base.

8.9.2 Protocolo de Meta-Objeto (MOP)

En esencia, un **MOP** [KRB92, GC96], es la meta-interfaz de implantación de un modelo de objetos. Dicho de otro modo, especifica la implantación de un modelo de objetos reflectivo.

Si se considera que el meta-nivel es una implantación del modelo de objetos soportado por el lenguaje, un MOP especifica **qué objetos del meta-nivel son**

necesarios para implantar el modelo de objetos. Y la interfaz exportada por cada uno de esos objetos forma parte de la interfaz del MOP [GC95, GC96].

En un sistema orientado a objetos donde, tanto las aplicaciones como el propio sistema, se modelan y representan en tiempo de ejecución como objetos que interaccionan entre sí, el meta-nivel especifica la implantación del modelo de objetos soportado por el sistema en términos de los meta-objetos necesarios para lograr el comportamiento deseado del sistema [KRB97].

Por ejemplo, serían necesarios meta-objetos para representar las clases, los métodos, los atributos de los objetos, con métodos que permitiesen consultar y modificar la signatura de un método, los atributos que componen el estado interno de un objeto, eliminando o insertando alguno nuevo. También podrían ser necesarios objetos que representasen la ejecución de métodos en los objetos permitiendo consultar y modificar el estado de cada una e incluso eliminar alguna.

El Meta-Protocolo o meta-interfaz se refiere, no a la especificación de la interfaz del objeto, sino a la especificación de la implantación de los objetos.

Entre los ejemplos más destacados cabe citar el MOP de CLOS [KRB91], o el MOP de CodA [McA93].

8.10 Reflectividad y sistemas operativos

Una aproximación novedosa en la arquitectura de un sistema operativo son las arquitecturas orientadas a objeto, donde cada componente del sistema operativo se define y encapsula como un objeto y, por tanto, el mecanismo de invocación entre objetos es el único mecanismo de interacción en esta arquitectura [TNO92].

8.10.1 Arquitectura reflectiva de un sistema operativo basado en meta-objetos

La aplicación de la reflectividad a la definición de la arquitectura del sistema operativo mediante la extensión de la aproximación orientada a objetos basada en meta-objetos [TNO92, KL93], supone que, si bien la funcionalidad se ofrecerá en forma de objetos o meta-objetos, existirá una separación clara entre unos y otros.

Mientras los primeros definen únicamente tareas de la aplicación a nivel usuario, el conjunto de meta-objetos define el comportamiento de los objetos de usuario, su entorno de ejecución. De esta forma, una llamada al sistema tradicional se transforma en una invocación a un método ofrecido por un meta-objeto [LYI95].

8.10.2 Separación objeto/meta-objeto

La computación de los objetos de usuario se gestionará mediante un conjunto de objetos, los meta-objetos, que definen el modelo de objetos de los primeros [TNO92].

Por ejemplo, en la arquitectura reflectiva definida en Apertos [Yok92] un grupo de meta-objetos forman un meta espacio y el punto de entrada al meta espacio se denomina reflector. El meta-espacio define el comportamiento de cada objeto de usuario y un objeto de usuario puede migrar entre diferentes meta-espacios que ofrezcan diferentes servicios manteniendo siempre la compatibilidad. Por ejemplo, si un objeto de usuario necesita moverse a almacenamiento secundario, el objeto debería migrar a un meta-espacio que dé soporte a la persistencia.

La diferencia entre el núcleo tradicional y la arquitectura reflectiva es que el enlace entre objetos y meta-objetos es dinámico de tal forma que la aplicación puede cambiar el comportamiento de los objetos cambiando su meta-espacio.

8.11 Resumen

La reflectividad es la capacidad de un programa de manipular, como si de datos se tratase, la representación del propio estado del programa durante su ejecución. En esta manipulación existen dos aspectos fundamentales: introspección e intervención. La introspección es la capacidad de un programa de observar y razonar acerca de su propio estado, sin posibilidad de manipularlo. La intervención es la capacidad de un programa de modificar su propio estado de ejecución o alterar su propia interpretación. Ambos aspectos requieren un mecanismo que codifique el estado de ejecución como datos. La exposición consiste en proporcionar tal codificación.

Un sistema reflectivo está compuesto por dos o más niveles que constituyen una torre reflectiva. Cada nivel sirve como nivel base para su nivel superior y como meta-nivel para su nivel inferior.

La reflectividad encaja especialmente bien con los principios promovidos por las tecnologías orientadas a objetos. Para introducir la reflectividad de manera uniforme en un sistema orientado a objetos es interesante que el modelo del meta-sistema esté descrito con el mismo paradigma orientado a objetos que el propio sistema base. Un objeto del nivel base es un objeto definido por la aplicación y que se refiere al ámbito del problema, mientras que un objeto del meta-nivel es un objeto que da soporte a los objetos del nivel base para lo que representan información acerca de los mismos y determinan su comportamiento.

Una parte fundamental de una arquitectura reflectiva es el mecanismo de transferencia de control desde el nivel-base al meta-nivel. Existen dos posibilidades: reflectividad implícita y reflectividad explícita.

El tipo de reflectividad especifica cuáles son los aspectos que se exponen y sobre los que se puede actuar. Se distinguen, en este caso, dos tipos: reflectividad estructural y reflectividad del comportamiento. A su vez, hay varios modelos de reflectividad que se pueden aplicar a los dos tipos anteriores: el modelo meta-clase, el modelo meta-objeto y la exposición de la comunicación.

Una aproximación novedosa en la construcción de sistemas operativos es la utilización de tecnologías orientadas a objetos. La aplicación de la reflectividad a la definición de su arquitectura mediante la utilización de meta-objetos permitirá ofrecer su funcionalidad por medio de estos, pero existiendo una separación clara con respecto a los objetos. Los objetos definirán tareas de las aplicaciones a nivel usuario, mientras que los meta-objetos definirán su comportamiento.

CAPÍTULO 9 EXTENSIÓN REFLECTIVA DE LA MÁQUINA ABSTRACTA PARA LA DISTRIBUCIÓN

9.1 Introducción

En este capítulo se describe la forma en que se va a organizar el sistema integral en base a la reflectividad, es decir, cómo van a cooperar máquina abstracta y sistema operativo con el objetivo último de dotar a este de la característica de distribución de los objetos.

Para ello, es necesario tomar un conjunto de decisiones de diseño referidas al modelo y tipos de reflectividad a implantar, basándose en las distintas alternativas que se describieron en el capítulo anterior. El resto del capítulo describe las decisiones adoptadas y la forma en que se interpretan de acuerdo a la arquitectura del sistema integral.

9.2 Arquitectura reflectiva propuesta

A continuación se describe el conjunto de decisiones adoptadas en lo relativo al modelo de reflectividad a implantar.

9.2.1 Torre reflectiva: dos niveles de objetos

Un sistema reflectivo OO, se organiza en dos o más niveles, que constituyen la torre reflectiva. Algunos sistemas, como Apertos [Yok92], tienen varios niveles de reflectividad lo que los convierte en sistemas complejos, difíciles de entender para usuarios poco avezados y con un bajo rendimiento.

La propuesta de esta investigación es adoptar una torre con únicamente **dos niveles**. Un **nivel base**, los objetos de la aplicación, y un **meta-nivel**, la máquina abstracta descrita por medio de objetos.

El meta-modelo de la máquina abstracta promueve representar mediante meta-objetos algunos aspectos de la misma haciéndolos de esta forma accesibles al nivel base. Así, es factible para el sistema operativo extender y modificar el funcionamiento de la máquina abstracta proporcionando sus propios objetos para que sobrescriban el comportamiento por defecto (los objetos por defecto) que esta proporciona a los objetos del nivel base. Gracias a esto, se consiguen especializar los aspectos que nos interesan de forma más sencilla y eficiente, fundamentalmente los relacionados con la distribución de objetos.

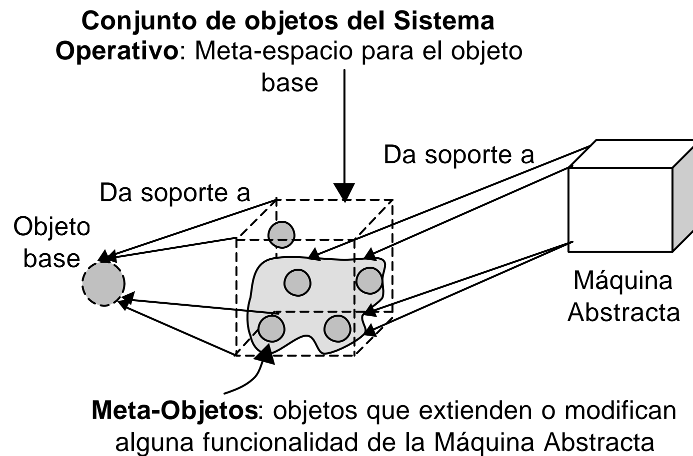


Figura 9.1. Dos niveles de objetos: el nivel base y el sistema operativo.

9.2.2 Modelo de reflectividad

Al diseñar un sistema reflectivo, una de las primeras preguntas que hay que responder es cómo se representa la reflectividad, o dicho de otra forma, cuál es el modelo de reflectividad elegido (en el capítulo anterior se hace una descripción de los modelos de reflectividad posibles).

9.2.2.1 El modelo de meta-objetos

La propuesta que aquí se hace para el meta-nivel supone representar la funcionalidad a exponer, el modelo, mediante objetos. O lo que es lo mismo, se representan las características estructurales y las características de ejecución de las aplicaciones en forma de objetos (**meta-objetos**) que componen el entorno computacional por omisión.

El uso de meta-objetos, además de permitir un poder de representación del entorno en tiempo de ejecución mayor que en otros casos, tiene varias características que lo hacen atractivo: la uniformidad y la flexibilidad.

Uniformidad

En primer lugar, el meta-objeto no rompe la uniformidad del sistema en torno al concepto único de objeto y el entorno en tiempo de ejecución se representa en forma de objetos con los que se trabaja mediante el uso de referencias y el paso de mensajes, como con cualquier otro objeto.

Flexibilidad

La especialización de los meta-objetos es posible gracias a que estos están asociados a objetos lo que hará posible la adaptación del entorno de ejecución de las aplicaciones ofreciendo a las mismas la posibilidad de cambiar la forma de representar los datos, estrategias de ejecución, mecanismos y protocolos [BG96].

Este entorno de computación flexible proporciona una representación de las características relevantes e importantes del sistema en términos del sistema mismo, al tiempo que evita el problema de parcialidad del que adolecía el modelo de Meta-Clase.

9.2.3 Descripción separada del nivel base y el meta-nivel

En este trabajo, al igual que en otros trabajos como Apertos [Yok92], se propone describir separadamente el objeto y su meta-nivel. De esta forma, los programadores pueden centrar su atención en describir los métodos que solucionan los problemas. Otros problemas, como encontrar el objeto destino o gestionar el envío de mensajes (o invocación de métodos), se describen en el meta-nivel.

Así, por ejemplo, un objeto del nivel base, puede describirse independientemente de su ámbito de actuación y gestión de recursos (local o remoto). Este aspecto de su ejecución, si puede solicitar recursos remotos o únicamente recursos locales y otros, se describe en el meta-nivel.

9.2.4 Utilización del paradigma de OO para describir el meta-nivel

Para introducir la reflectividad en el sistema de manera uniforme, lo más interesante es que el modelo del **meta-sistema esté descrito utilizando el mismo paradigma OO** que el propio sistema base. De este modo, tanto el modelo, como las operaciones de exposición y reflexión pueden ser utilizados de manera uniforme dentro del mismo paradigma.

Así, el **nivel base** estará compuesto por los objetos que definen la aplicación (lo que hace el objeto), mientras el **meta-nivel** estará compuesto por otros objetos (**uniformidad**) que definen el entorno de ejecución de los objetos del nivel base o de aplicación.

La clave en el diseño de la arquitectura del meta-nivel es la **descomposición** o **factorización** del mismo en objetos de grano fino **o meta-componentes** o **meta-objetos**. De esta forma se mantiene la **uniformidad conceptual en torno al paradigma de la orientación a objetos** en todo el sistema.

Se puede establecer una analogía clara con la programación tradicional. Cuando se desarrollan o programan sistemas orientados a objeto del nivel base o nivel de la aplicación, se descompone el comportamiento en piezas pequeñas y manejables, se crean objetos para cada pieza y se componen tales objetos en aplicaciones.

La única diferencia aquí es que la aplicación es el modelo de objetos que describe el funcionamiento de los objetos de la aplicación de usuario. Es decir, el meta-nivel es una aplicación cuyo dominio es el comportamiento de los objetos, cómo se comportan.

Al descomponer el meta-nivel en meta-componentes, se desarrolla un **modelo de ejecución de objetos relativamente genérico**. La descripción del meta-nivel se hace en función del papel que juega en la descripción del comportamiento del objeto del nivel base.

Un meta-componente o meta-objeto se encargará de cumplir cada uno de los papeles que, normalmente se corresponderá con parte del comportamiento de un objeto como: ejecución de objetos (tanto los mecanismos como los recursos), paso de mensajes, correspondencia de mensajes con métodos, y otros.

Cada papel a representar puede ser interpretado por varios meta-objetos diferentes y, algunas veces, un meta-objeto puede jugar varios papeles.

El comportamiento de un objeto cambia por medio de la redefinición explícita de componentes o bien, extendiendo el conjunto de papeles.

9.2.5 El nivel base

Los objetos del nivel base están formados por datos y métodos que modelan una entidad del problema y que interaccionan entre sí gracias a la invocación de métodos, lo que se traduce en paso de mensajes entre objetos. Este es el único mecanismo ofrecido para comunicar objetos y proporciona además los principios básicos para la sincronización de los mismos.

La máquina abstracta ofrece el objeto como entidad autónoma de procesamiento (modelo de objetos activo), sin ofrecer ningún mecanismo de gestión de la ejecución de sus métodos. Por tanto, los mecanismos relacionados con la distribución de objetos, como por ejemplo la invocación remota de métodos, tendrán que ser proporcionados en el meta-nivel.

9.2.6 El meta-nivel

Como se ha visto en el apartado anterior, el objeto base, tal como está definido, no está dotado de los mecanismos necesarios que le permitan ofrecer la semántica que se le exige para la distribución. Por tanto, para poder ofrecer este comportamiento, es necesario complementar el objeto base con un meta-espacio que dota al mismo de las capacidades adicionales necesarias.

La propuesta que se hace para el meta-nivel supone exponer ciertos aspectos de la máquina mediante un conjunto de meta-objetos.

Es decir, se trata de **asociar a cada objeto un meta-espacio**, compuesto por uno o varios meta-objetos que expongan distintos aspectos del entorno en tiempo de ejecución del objeto base (meta-modelo de la máquina abstracta). Esta asociación es dinámica en tiempo de ejecución, puede eliminarse o cambiarse en cualquier momento y permite cambiar la semántica del objeto dinámicamente, lo que confiere gran flexibilidad al entorno.

Los meta-objetos por omisión serán proporcionados por la máquina abstracta. El sistema operativo pone a disposición de los usuarios meta-objetos más especializados para los aspectos expuestos.

9.2.6.1 La máquina abstracta como conjunto de objetos

Aunque la máquina ofrezca una funcionalidad mínima, esta incluye, típicamente, partes fundamentales del comportamiento del entorno como por ejemplo la comunicación entre objetos.

Por tanto, en aras de la tan deseada flexibilidad del entorno, será necesario actuar sobre ella de forma que su comportamiento sea accesible y modificable para los objetos de usuario.

Elevación de los objetos de la máquina abstracta OO

Para que el modelo de objetos y la funcionalidad implantados en la máquina abstracta pueda ser modificable para las aplicaciones, debe describirse de forma que constituya una entidad para la aplicación que quiera modificarla, es decir, **el modelo de objetos debe codificarse como un conjunto de objetos**.

Dotar a la máquina abstracta orientada a objetos de una **arquitectura reflectiva** supone que parte o toda la **funcionalidad de la máquina abstracta**, además de implementarse internamente en la misma, se ofrece en un nivel superior, **como un**

conjunto de objetos que componen el **modelo de la máquina**, sobre los que se puede actuar para cambiar el comportamiento de la misma.

Es decir, **el modelo de objetos debe exponerse a los objetos de usuario como un conjunto de objetos normales**, que forman parte del mundo de objetos que define el sistema integral.

De esta forma, la funcionalidad pasa de estar codificada internamente en la máquina, en el lenguaje de desarrollo elegido para ello, y por tanto ser totalmente intocable, a describirse como objetos, codificados en el lenguaje definido por la propia máquina, de forma que constituya una entidad visible para aquella aplicación que quiera modificarla.

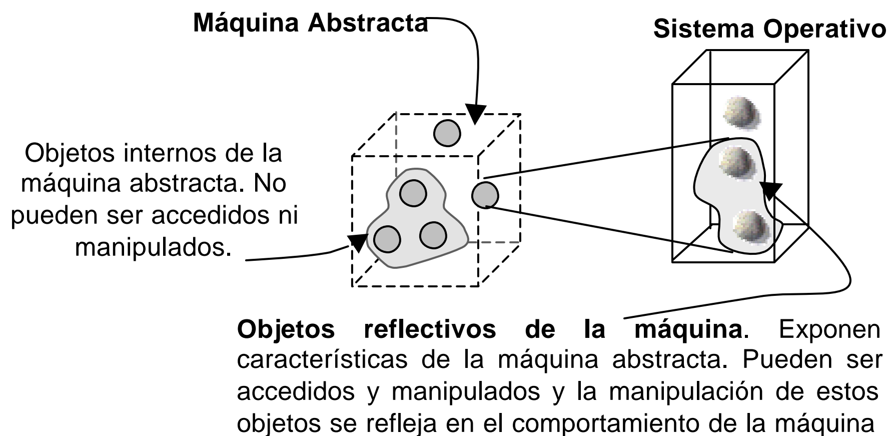


Figura 9.2. Modelo de objetos de la máquina codificado como objetos reflectivos

9.2.6.2 El sistema operativo como conjunto de objetos

La máquina abstracta OO, aunque soporta en parte la abstracción de objeto y los mecanismos que permiten comunicarlos y ejecutar sus métodos, no es suficiente para definir completamente un entorno distribuido OO.

El **resto de la funcionalidad** necesaria la proporciona el sistema operativo a través de una serie de objetos que definen parcelas de la distribución de objetos como la invocación remota y la migración. Así, las llamadas al sistema de los sistemas operativos tradicionales se traducen en este entorno por invocaciones a métodos de objetos del sistema operativo.

De esta forma, el **sistema operativo** se integra en el entorno de forma homogénea, ya que también ofrece su funcionalidad como un **conjunto de objetos** que extienden, modifican o complementan el comportamiento de la máquina [TAD+98b].

9.3 Reflectividad estructural

Para poder lograr la modificación de la máquina abstracta en tiempo de ejecución y permitir de esta forma la cooperación entre máquina abstracta y sistema operativo, es preciso modificar en algunos aspectos la arquitectura interna de la máquina, dotándola, inicialmente, de **reflectividad estructural**.

Se trata, fundamentalmente, de convertir los objetos del sistema en objetos visibles y manipulables por el usuario. Para ello se definen, en el lenguaje definido por la máquina abstracta, un conjunto de clases básicas que forman el módulo de reflectividad

estructural de la máquina [Fer99]. Se trata del área de clases, el área de instancias, el área de referencias, el área de hilos, las clases, las instancias, los métodos y las instrucciones, que se convierten en objetos manipulables en tiempo de ejecución.

Esta jerarquía de clases básicas exponen o reflejan, aspectos de implantación de la arquitectura de la propia máquina. A partir de estas clases, la información elegida se mantendrá en el meta-nivel del objeto concreto, representada como objetos de primera clase en tiempo de ejecución, con una serie de métodos que podrán ser invocados dinámicamente, aunque siempre de forma síncrona, y que provocarán la modificación de dichos elementos en tiempo de ejecución, según las necesidades de la aplicación.

La gestión y manipulación de estos objetos se lleva a cabo por distintos meta-objetos que componen el entorno del objeto base, pudiendo aplicarse sobre ellos la característica de **introspección** (poder saber qué, cómo y por dónde se está ejecutando, qué está siendo retrasado su ejecución, etc.) e **intervención**, al poder modificarse dichos métodos. Esta última propiedad es fundamental para la consecución de la **reflectividad del comportamiento** o adecuación del entorno dinámicamente.

Se trata, en ese caso, de exponer la composición de las instancias en tiempo de ejecución, de forma que no sólo se pueda acceder a la información de los mismos sino que incluso se puedan manipular las instancias dinámicamente.

9.3.1 Exposición de la arquitectura interna de la máquina abstracta

La reflectividad estructural consiste en exponer la propia arquitectura de la máquina, permitiéndose a los objetos acceder a su funcionalidad interna en tiempo de ejecución como si estuviesen accediendo a otro objeto cualquiera.

9.3.1.1 Área de clases

La exposición del área de clases de la máquina como un objeto tiene como finalidad permitir el acceso y modificación del área de clases de la máquina.

Gracias a este objeto, es posible averiguar qué clases contiene el área de clases, es decir, qué clases hay definidas en un momento determinado (introspección), así como hacer visibles y manipulables tales clases gracias a la exposición del área de clases y de las propias clases.

Es el primer paso para poder crear clases de usuario en tiempo de ejecución y con ello modificar la jerarquía de clases dinámicamente. Esto, junto con la exposición de instancias y referencias, es fundamental para la **consecución de la reflectividad del comportamiento.**

9.3.1.2 Área de instancias

La exposición del área de instancias de la máquina como un objeto tiene como finalidad permitir el acceso y modificación del área de instancias de la máquina.

Igual que en el caso anterior, el objeto que representa al área de instancias en tiempo de ejecución permite conocer qué instancias hay creadas en tiempo de ejecución (introspección), gracias a lo cual, junto con la exposición de las propias instancias, los propios objetos son visibles y manipulables en tiempo de ejecución. Por otro lado, permite también crear instancias de una clase en tiempo de ejecución.

9.3.1.3 Área de referencias

El área de referencias no existe como tal en la máquina abstracta. Es un área virtual que sirve únicamente para establecer la correspondencia entre la referencia a un objeto y su identificador dentro de la máquina. Sin embargo, para cumplir un papel similar se exponen las referencias de los objetos.

Esto permite el acceso y creación de nuevas referencias en la máquina abstracta en tiempo de ejecución. La creación de nuevas referencias en tiempo de ejecución es fundamental para poder definir clases en tiempo de ejecución: los agregados, asociados, ancestros de una clase, argumentos de un método, etc.

Permite conocer el nombre de una referencia, la clase a la que pertenece o si está libre o instanciada.

9.3.1.4 Área de hilos

Por último, la exposición del área de hilos del sistema permite el acceso al área de hilos de la máquina abstracta.

Esto permite conocer los hilos que hay en el sistema, tanto el número de ellos como obtener los hilos propiamente dichos.

9.3.2 Exposición de la implantación en tiempo de ejecución de los objetos

9.3.2.1 Las clases

La exposición de las clases tiene una doble finalidad. En primer lugar, permitir el acceso a las clases en tiempo de ejecución así como poder realizar introspección sobre las mismas, conociendo su composición, sus métodos, la cadena de herencias, los agregados y asociados, etc.

Por otro lado, permite la creación de clases en tiempo de ejecución, definiendo completamente sus métodos, agregados, etc.

Esto, junto con la exposición del área de clases permitirá añadir nuevas clases al área de clases en tiempo de ejecución, ampliando la jerarquía de clases de forma dinámica.

9.3.2.2 Las instancias

La exposición de las instancias, es decir, de los objetos en tiempo de ejecución, permite el acceso a las instancias de la máquina abstracta.

Gracias a la exposición de las instancias, la máquina permite conocer la composición de las instancias que hay creadas, permitiendo el acceso a sus agregados y asociados, la clase a la que pertenece, etc.

9.3.2.3 Los métodos

Si la exposición del área de clases permitía obtener los métodos que componen la clase, la exposición de los métodos en sí, permite el acceso y modificación de los mismos en tiempo de ejecución.

De esta forma se permite, no sólo crear nuevos métodos en tiempo de ejecución, imprescindible para definir nuevas clases dinámicamente, sino también realizar introspección sobre los métodos, conociendo sus argumentos, variables locales, instrucciones que lo componen, tipo del valor de retorno, etc.

9.3.2.4 Las instrucciones

Esta clase, que exponen las instrucciones de la máquina, se crea fundamentalmente, para permitir la definición de clases en tiempo de ejecución, permitiendo dotar de instrucciones a sus métodos.

Fundamentalmente, permite crear nuevas instrucciones, de entre el conjunto de instrucciones de la máquina, en tiempo de ejecución que luego serán asignadas a algún método.

También permite ciertas capacidades de introspección, al permitir averiguar características sobre las instrucciones, como por ejemplo de qué instrucción se trata.

9.3.2.5 Hilos de ejecución

La exposición de los hilos permite el acceso y modificación del propio hilo, es decir, de la ejecución de un método.

Ofrece la posibilidad de conocer qué método se está ejecutando, obtener el estado del hilo, las referencias a las instancias locales, así como conocer la cadena de llamadas que han llevado hasta la situación actual.

9.3.3 Exposición del meta-espacio de los objetos

Además de la exposición de la arquitectura interna de la máquina, es necesario también exponer la composición de las instancias en tiempo de ejecución, es decir, la implantación de los objetos en tiempo de ejecución, de forma que no sólo se pueda acceder a la información de los mismos en tiempo de ejecución (**introspección**), sino que incluso, se puedan manipular las instancias dinámicamente (**intervención**).

De la misma forma que antes, para lograrlo, se exponen los distintos elementos que constituyen las instancias y que, a partir de este momento, podrán ser creados y manipulados en tiempo de ejecución.

9.3.3.1 El meta-espacio de un objeto

Un objeto pertenece a un meta-espacio en tiempo de ejecución, que señala el comportamiento de la máquina abstracta en determinados aspectos, para ese objeto. Un meta-espacio (ver capítulo anterior), no es más que un conjunto de objetos, cuyos métodos modifican el comportamiento de la máquina abstracta.

Un objeto puede determinar un comportamiento por omisión de la máquina y, sin embargo, puede desear cambiarlo, en función de su estado y su entorno en cualquier momento, en tiempo de ejecución.

La exposición del meta-espacio permite el acceso y manipulación del mismo lo que permite dotar a la máquina de un comportamiento distinto en tiempo de ejecución.

Permite interrogar al objeto acerca de los aspectos expuestos – aquellos que tendrán asociado un meta-objeto – y conseguir las referencias a los distintos meta-objetos.

Junto con la exposición de clases, permite definir nuevos comportamientos – nuevos meta-objetos – en tiempo de ejecución, y, unido a la exposición de instancias y del meta-espacio, permite modificar, en tiempo de ejecución, el meta-espacio de un objeto.

9.3.3.2 Los meta-objetos

Permite el acceso y modificación en tiempo de ejecución de un meta-objeto. De esta forma, se puede interrogar a un meta-objeto acerca del aspecto de la máquina que

expone, de sus métodos o lo que es lo mismo, cómo modifica el comportamiento de la máquina, etc.

Igualmente, y gracias a la exposición de métodos e instrucciones, será posible modificar un meta-objeto en tiempo de ejecución, permitiendo de esta forma variar su comportamiento dinámicamente.

9.3.4 La reflectividad estructural en la distribución de objetos

La reflectividad estructural es imprescindible para los mecanismos de invocación remota de métodos y migración de objetos. Incluso algunas políticas asociadas a los mecanismos citados pueden sacar partido de la posibilidad de acceder a información estructural de las máquinas abstractas y de los objetos en tiempo de ejecución.

En los próximos capítulos se describe en detalle cómo es utilizada la reflectividad estructural en el diseño de los mecanismos de la distribución de objetos. A continuación se muestran algunos ejemplos que sirven, de momento, para comprender la necesidad de este tipo de reflectividad.

Una operación de invocación necesita, en primer lugar, conocer la ubicación del objeto invocado. Dado que un objeto debe residir en el área de instancias de alguna máquina abstracta, lo habitual es comenzar la búsqueda en el área de instancias de la máquina abstracta en la que se produce la invocación. El **objeto reflejado del área de instancias** proporcionará un método para realizar en ella la búsqueda de un objeto, conocido su identificador.

Dado que un objeto va a poder migrar por los diferentes nodos que componen el sistema integral distribuido, puede presentarse el caso de que un objeto no esté ubicado en la misma máquina abstracta que almacena la descripción de su clase. Si el objeto es invocado, será necesario obtener una copia del método a ejecutar. La copia del método se conseguirá invocando el método apropiado del **objeto reflejado del área de clases** de la máquina abstracta en la que está contenida la descripción de la clase. Así mismo, en la propia operación de obtención del método será necesario manipular **objetos que representen al mismo y a sus instrucciones**.

El conocimiento de la carga de cada una de las máquinas abstractas del sistema integral distribuido permitiría la implantación de mecanismos y políticas para el equilibrado de carga. Esto es posible por la existencia de **objetos que reflejan las áreas de hilos** de las diferentes máquinas abstractas, de modo que se puede conocer, con simples invocaciones a métodos de dichos objetos, la carga de cada máquina para tomar decisiones de migración de objetos que equilibren la carga de todas ellas.

En cuanto a la exposición de la implantación de los objetos en tiempo de ejecución, en los próximos capítulos se mostrará la necesidad de mantener meta-información de cada uno de ellos, en lo relativo a la distribución. Toda esa información se almacena en un **meta-objeto datos**, privado para cada objeto. Por ejemplo, un objeto será susceptible de moverse si el atributo `móvil` almacenado en su meta-objeto datos tiene un valor lógico `verdadero`.

9.4 Reflectividad del comportamiento

Para lograr el objetivo de extender la máquina adaptando su comportamiento a las aplicaciones que se ejecutan sobre ella, es la reflectividad de comportamiento el campo que mejor lo permite.

Dado que la reflectividad encaja especialmente bien con los conceptos de objeto y OO, una extensión natural de la exposición de la estructura consiste en organizar el control del comportamiento de un sistema de computación OO (su meta-interfaz) a través de un conjunto de objetos denominados meta-objetos [BG96].

Se trata de exponer el comportamiento de la máquina (su forma de ejecutar las aplicaciones), para lo que es necesario cosificar los elementos de la máquina que hacen que funcionen los objetos del nivel base.

Los meta-objetos pueden representar varias características del entorno de ejecución, como la representación de la ejecución, la ejecución en sí, la comunicación o la localización, de tal forma que, la especialización de estos meta-objetos puede extender y modificar el contexto de ejecución de un conjunto específico de objetos del nivel base.

La reflectividad del comportamiento también puede expresar y controlar la gestión de recursos, no a nivel de un objeto individual, sino a un nivel mucho más amplio como la planificación, el agrupamiento de objetos, etc.

Esto ayuda a ejercer un control de grano mucho más fino, por ejemplo para planificación y equilibrado de carga, en oposición a algún algoritmo global y fijo que, normalmente, se optimiza para alguna clase de aplicación.

9.4.1 Aspectos de la máquina que migran inicialmente al meta-nivel

Los objetivos del meta-nivel referidos al entorno de computación de los objetos, se refieren, fundamentalmente, a los mecanismos que es necesario proporcionar para permitir la implantación de la distribución de objetos.

Como **base de la arquitectura reflectiva para la distribución de objetos**, se propone en esta tesis tomar en consideración los aspectos relacionados con la comunicación entre objetos, que dará lugar a un conjunto de meta-objetos.

En [Alv98] y [Taj2000] se discuten en profundidad otros aspectos relacionados, respectivamente, con la persistencia y la computación del sistema. De todas maneras, y con el fin de proporcionar una visión más completa, se describe de manera escueta de qué manera se proporciona el control de la sincronización intra-objeto y entre objetos.

9.4.1.1 Comunicación entre objetos

La exposición del envío y recepción de mensajes, o lo que es lo mismo, la **exposición del comportamiento de la máquina para la invocación de métodos**, permite a los objetos del nivel base invocador e invocado establecer protocolos privados de comunicación, mecanismos de búsqueda del objeto destino, definir un comportamiento específico ante errores o condiciones excepcionales o instalar temporizadores para establecer tiempos de respuesta.

Con el fin de proporcionar la **máxima flexibilidad**, se expone el envío de mensajes tanto en el extremo emisor (objeto invocador) como en el extremo receptor (objeto invocado).

El meta-componente **mensajero** se compondrá entonces de dos meta-objetos, virtualmente independientes, cada uno de los cuales se encarga de uno de los aspectos de la gestión de mensajes, el envío o la recepción, y que dialogan mediante el paso de mensajes para llevar a cabo la invocación efectiva del método.

Estos dos meta-objetos recibirán el nombre de **mensajero-emisor** o emisor y **mensajero-receptor** o receptor para indicar, no sólo su funcionalidad sino también el origen del meta-objeto.

9.4.1.2 Control de la actividad interna de los objetos

Un modo de introducir el control de la concurrencia en el sistema es separar la descripción de la sincronización del código secuencial de tal modo que el código de sincronización no se mezcle con el código de los métodos y se consiga una separación más clara de algoritmo y sincronización. La idea básica del mecanismo de sincronización compartido con [Rei97, CV98, VA98] es dividir el estado del objeto concurrente en parte secuencial y parte concurrente como se muestra en la figura siguiente.

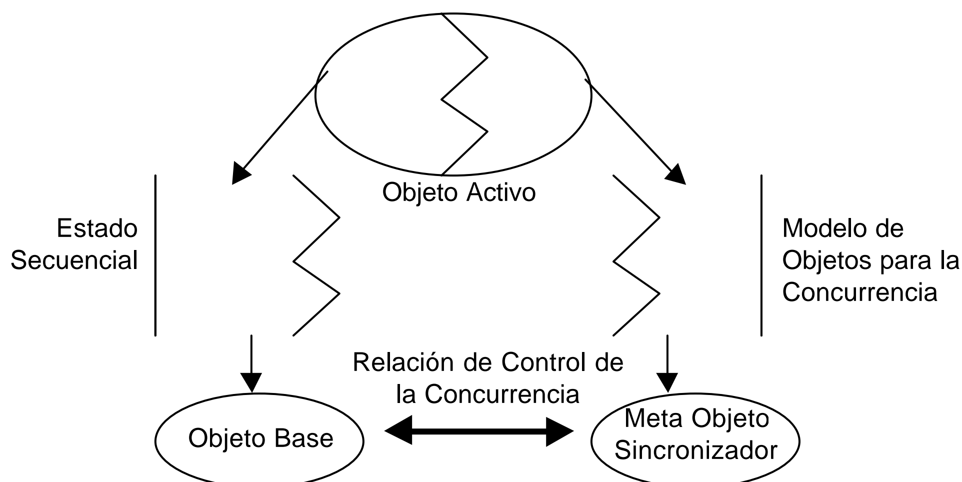


Figura 9.3. Control de la concurrencia en el meta-espacio.

La parte secuencial contiene el estado y los métodos que implementan el comportamiento del código funcional. Este es el objeto base. Por su parte, el objeto concurrente contiene el código no funcional, relativo a aspectos de sincronización, es decir, el estado y los métodos que implementan el comportamiento del objeto de cara a sincronizar la ejecución concurrente de varios métodos del objeto base. Este es el meta-objeto denominado **sincronizador**.

Así, si el meta-objeto receptor definía el protocolo de recepción de mensajes, el meta-objeto sincronizador define la **política de aceptación de mensajes**, lo que significa que determina cómo y cuándo se ejecutan los distintos métodos de un objeto.

Para ello, cuando el meta-objeto receptor recibe un mensaje correspondiente a una invocación, cede el control al meta-objeto sincronizador para determinar la posible aceptación del mensaje.

El meta-objeto sincronizador consulta la especificación del comportamiento del objeto definida por el usuario (o se guía por el comportamiento por omisión) estudiando las distintas restricciones de ejecución antes de determinar si la ejecución de ese método es adecuada o no.

Esto dependerá del entorno de ejecución actual del objeto: qué métodos ya están en ejecución (lo que no implica que tengan asociado en este preciso instante un procesador físico), qué métodos están a la espera de ejecutarse y qué métodos están parados en alguna condición de sincronización.

9.4.1.3 Sincronizacion entre objetos

Otro requisito funcional relevante en un entorno de ejecución concurrente es permitir la **concurrencia externa**, es decir, la posibilidad de que varios objetos compitan por los recursos necesarios para ejecutar sus métodos, en concreto, por el control de la máquina subyacente.

De todo esto se deriva la necesidad de establecer una **política de planificación** que determine el orden de ejecución de las tareas tanto internas como externas.

El **meta-objeto planificador** es el responsable de la planificación de las actividades de un objeto o un conjunto de ellos. En general, un conjunto de objetos del nivel base comparten un único meta-objeto planificador al que están asociados. Dado que los meta-objetos planificadores son a su vez objetos, será necesario planificar su ejecución con la utilización de otros meta-objetos. Se crea así una jerarquía de planificadores que se pasan el control unos a otros.

Los meta-objetos planificadores asociados a objetos del nivel base tendrán asociada una lista de hilos de ejecución, de uno o varios objetos, que pelean por el control de la máquina para ejecutar las instrucciones del método al que están ligados. Estos planificadores obtendrán el control de algún planificador jerárquicamente superior, ejecutando entonces las acciones determinadas por la política de planificación que implementa para elegir, entre todos los hilos que él controla, el más adecuado para su ejecución.

Estos meta-objetos también son invocados por otras causas. La más común es cuando el meta-objeto sincronizador decide que se dan las condiciones necesarias para que la ejecución de un método invocado del objeto base se lleve a efecto. En este caso, el meta-objeto sincronizador solicita al meta-objeto planificador que encole el hilo para que pueda ser planificado en algún momento.

9.4.2 La reflectividad del comportamiento en la distribución de objetos

La reflectividad del comportamiento descrita se sustenta en gran medida sobre el mecanismo de paso de mensajes definido por la máquina abstracta. Dado que los meta-objetos son objetos de primera clase, debe ser factible comunicarse con ellos vía paso de mensajes.

La invocación de un objeto del nivel base por parte de otro objeto del nivel base provoca el paso del control al meta-nivel del objeto. Con otras palabras, **la semántica del paso de mensajes definida en la máquina es sobrescrita** por el meta-objeto correspondiente.

No existe, en cambio una cesión del control al meta-nivel del objeto en los casos siguientes: cuando un meta-objeto invoca a otro meta-objeto y cuando un meta-objeto invoca a un objeto del nivel base. En ambos casos, la máquina abstracta actúa de meta-espacio básico, finalizando así la recursión reflectiva, ya que este es el último paso para la ejecución efectiva del método.

Los **meta-objetos emisor y receptor** se encargan de los aspectos relacionados con el paso de mensajes en el meta-nivel. En los apartados siguientes se describe su funcionalidad y de qué manera deben ser utilizados para permitir implantar un mecanismo de invocación de objetos que sustituya al que por defecto proporciona la máquina abstracta, con el fin de lograr la invocación remota de objetos de modo transparente.

9.4.2.1 El meta-objeto mensajero-emisor

El meta-objeto emisor se encarga de realizar las **acciones que el nivel base disponga para la invocación del método** o envío del mensaje. Para ello, cuando un objeto del nivel base invoca un método de algún otro objeto del nivel base, o lo que es lo mismo, le envía un mensaje, el meta-objeto emisor del objeto origen de la llamada, captura el envío y toma el control de la operación de envío. En otras palabras, la máquina abstracta, en el **proceso de interpretación de la instrucción** invoca al meta-objeto emisor.

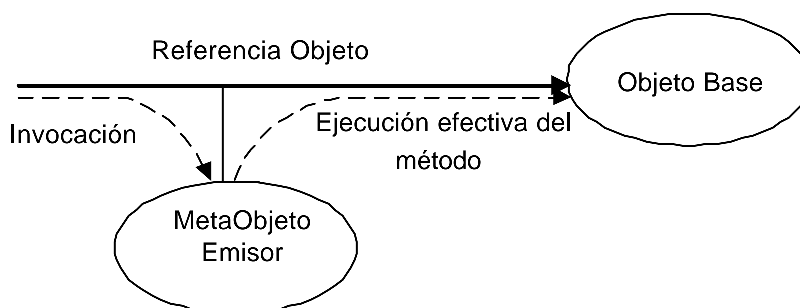


Figura 9.4. Invocación de métodos. Salto al meta-objeto emisor

Este meta-objeto expone diversos aspectos del envío de mensajes que el objeto de nivel base puede adaptar a su entorno de ejecución, de tal forma que es posible personalizar las operaciones que este meta-objeto lleva a cabo tanto antes como después de realizar el envío.

En general, dado que los meta-objetos se codifican en el lenguaje de la máquina, se pueden implantar sus métodos de tal forma que su comportamiento quede totalmente adaptado a los requerimientos de las aplicaciones.

Objetivo del meta-objeto

El principal objetivo del sistema de invocación de métodos, máquina y meta-objeto, es conseguir que el método invocado se ejecute como se especifica en el objeto del nivel base. Sin embargo, para dotar de flexibilidad al entorno, permite adaptar las acciones de invocación del método al entorno y objetos concretos de que se trate. Concretamente, **el emisor podrá especializar su comportamiento** para invocar un método en otro objeto, con el fin de implementar la semántica de invocación deseada.

Para ello, el meta-objeto puede hacer varias operaciones antes y después de que la operación de envío se lleve a cabo de forma efectiva, es decir, antes y después de que se cree un nuevo hilo en el objeto destino para servir la invocación.

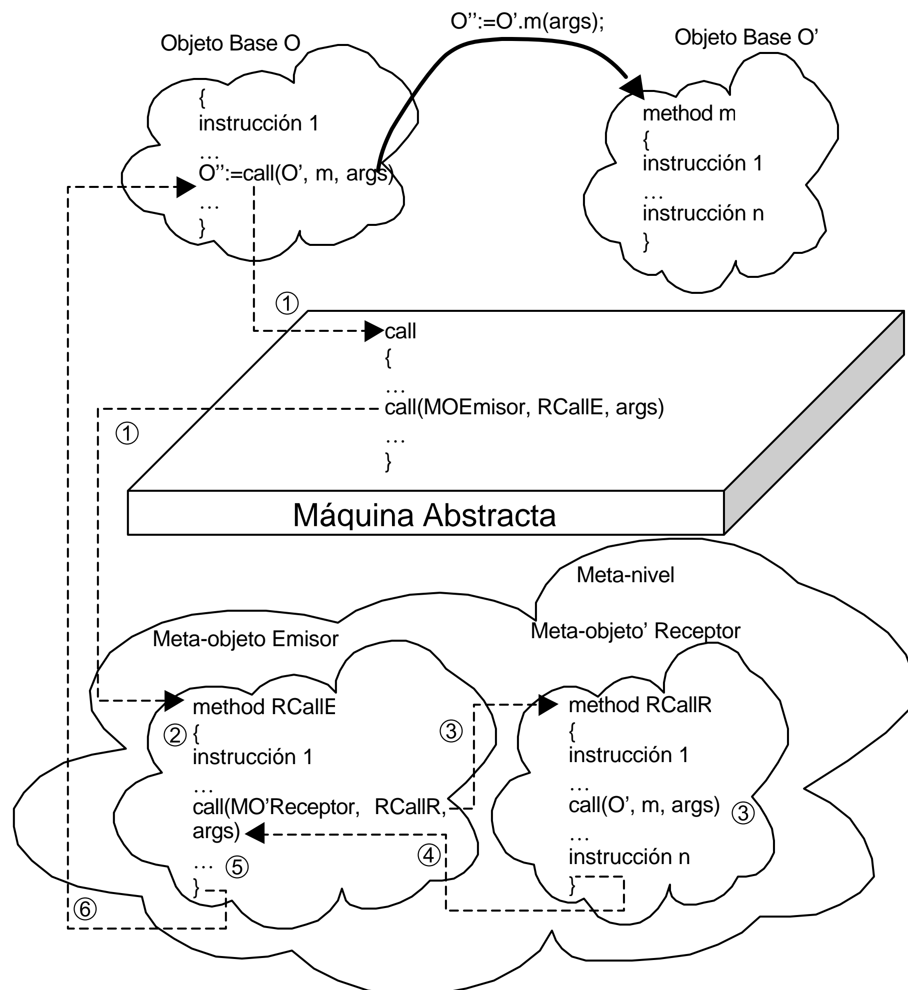
Comportamiento del meta-objeto

A continuación se muestra una aproximación más detallada del comportamiento del emisor de mensajes. Estas acciones tienen lugar cuando un objeto del nivel base invoca un método de otro objeto también del nivel `base`. En toda la descripción, se supone una invocación síncrona del método.

Envío de mensajes: ejecución de la instrucción de invocación por parte de un objeto

1. Transición del nivel base al meta-nivel. El control se pasa al meta-objeto emisor del objeto correspondiente mediante la invocación de su método `RCallE`.

2. Se ejecutan las acciones del meta-nivel señaladas por el meta-objeto emisor del lado origen del envío. Estas acciones son anteriores al envío efectivo del mensaje.
3. Avisar al objeto receptor mediante la invocación de su meta-objeto receptor (`RCallR`). Esta acción equivaldría a poner el mensaje en la cola del mensajero del receptor. Como las invocaciones a y entre meta-objetos son síncronas, el meta objeto emisor del origen se queda esperando la respuesta del meta objeto receptor del destino.
4. Cuando terminen las acciones del lado destino, el meta objeto emisor recupera el control. Esto sucede cuando termina la ejecución efectiva del método solicitado.
5. Una vez que recupera el control, el meta-objeto emisor puede realizar acciones posteriores a la invocación efectiva del método.
6. Finalmente, devuelve el control al objeto base, es decir, se produce una transferencia de control del meta nivel al nivel base.



① Transferencia del nivel base al meta-nivel

② Ejecución síncrona del método RCallE en el meta-objeto Emisor para la transmisión del mensaje. Se suspende la ejecución del método origen

③ Invocación síncrona del método RCallR del meta-objeto Receptor, por tanto, se suspende la ejecución del método RCallE. Supone la invocación efectiva del método m

④ Reanudación del método RCallE

⑤ Finalización del método RCallE

⑥ Transferencia de control del meta-nivel al nivel base. Reanudación del método origen.

Figura 9.5. Ejecución del meta-objeto emisor.

La única instrucción que se ejecutará en el objeto base será la instrucción de invocación que, inmediatamente, efectuará una transición al meta-nivel y será el meta-objeto emisor el que continúe.

En la implantación final el meta objeto emisor, al igual que el receptor y cualquier otro meta-objeto que sea necesario introducir, se escribirá en el lenguaje definido por la máquina abstracta, lo que implica que se puede definir en cualquier lenguaje de alto nivel para el que se disponga de un compilador a tal lenguaje ensamblador.

El meta-objeto emisor en la invocación remota de métodos

La descripción del funcionamiento del meta-objeto emisor es básicamente válida, tanto para una invocación local como para una invocación remota. La diferencia entre una invocación local y una remota vendrían dadas por:

- El conjunto de acciones a realizar por parte del meta-objeto emisor antes de la operación de envío del mensaje.
- La manera en que se va a conseguir el envío del mensaje al meta-objeto receptor del objeto invocado.
- El conjunto de acciones a realizar por parte del meta-objeto emisor una vez finalizada la operación de invocación propiamente dicha.

Para ser precisos, la clasificación de una invocación como local o remota se va a producir en la realización del primero de los puntos anteriores. Lo primero que tiene que hacer el meta-objeto emisor es localizar el objeto invocado: en la misma máquina abstracta que el objeto invocador o en una máquina abstracta distinta. Para saber si el objeto invocado está en la máquina local, el meta-objeto emisor hará uso del objeto reflejado del área de instancias local, al cual realizará la consulta pertinente. En caso de que no se encuentre en el área de instancias local, solicitará la ayuda del sistema de localización del sistema operativo, el cual buscará el objeto en el resto del sistema integral y devolverá su ubicación.

El envío del mensaje al meta-objeto receptor del objeto invocado va a depender del resultado del paso anterior. Si la ubicación de los objetos invocador e invocado es la misma, el envío del mensaje será satisfecho por la propia máquina abstracta. En el caso de que no coincidan sus ubicaciones, un conjunto adicional de objetos del sistema operativo se encargarán de hacer llegar el mensaje a la máquina abstracta en la que está ubicada el objeto invocado.

Finalmente, el meta-objeto emisor podrá realizar una serie de acciones una vez finalizada la invocación, como por ejemplo recolección de información de tipo estadístico sobre el tiempo consumido en la invocación remota.

De alguna manera, **se puede ver al meta-objeto emisor como un representante** (*proxy*) o sustituto (*stub*) de los sistemas distribuidos vistos en el capítulo 4. Sin embargo, los representantes actuaban únicamente para las invocaciones remotas, mientras que el meta-objeto emisor lo hace para todas aquellas invocaciones (locales o remotas) realizadas por objetos que tengan asociado este meta-objeto. Además, la **semántica de un meta-objeto emisor puede ser todo lo compleja que se desee**, y, como se verá, su papel puede ir mucho más lejos que la simple invocación de métodos de objetos remotos.

El conjunto de actividades a desarrollar por el meta-objeto emisor aparecen aquí muy resumidas y se concretan en un conjunto de mecanismos y políticas que forman parte del sistema de distribución del sistema integral. Dichos mecanismos y políticas se describen en profundidad en el capítulo 12.

9.4.2.2 El meta-objeto mensajero-receptor

El meta-objeto **receptor** se encarga de realizar las acciones que el nivel base disponga cuando reciba una invocación a uno de sus métodos o **llegada de un mensaje**. Para ello, cuando algún objeto del nivel base invoca un método de otro objeto del nivel base, o lo que es lo mismo, le envía un mensaje, el meta-objeto emisor virtualmente invoca el método correspondiente en el meta-objeto receptor del objeto destino de la llamada, que toma el control de la invocación y realiza diversas acciones antes de pasar el control al nivel base para la ejecución efectiva del método.

Este meta-objeto expone diversos aspectos de la recepción y ejecución de métodos que el objeto de nivel base puede adaptar a su entorno de ejecución, personalizando las operaciones que este meta-objeto lleva a cabo.

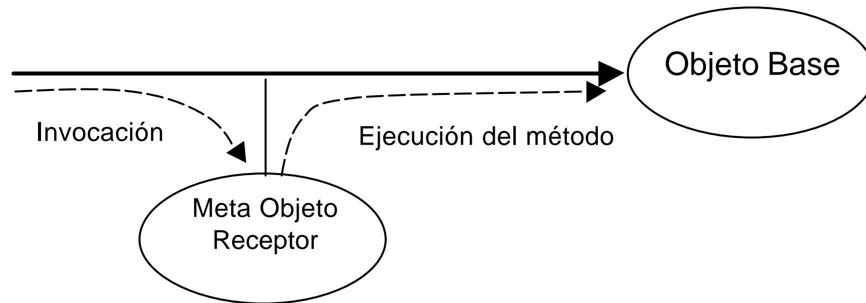


Figura 9.6. Invocación de métodos. Salto al meta-objeto receptor.

Ejecución del meta-objeto receptor

La recepción de mensajes en el lado receptor se produce como consecuencia de la invocación del método `RCallR` (reflejo de la parte receptora de la instrucción de invocación) del meta-objeto receptor por parte del meta-objeto emisor. La invocación de este método provoca que el meta-objeto receptor reciba los mensajes y, a medida que estos mensajes llegan, el receptor los acepta y analiza el contexto de ejecución del objeto para determinar la actuación a llevar a cabo.

Virtualmente, estos mensajes se colocan en una cola de mensajes (buzón) del meta-objeto receptor asociado al objeto destino. El meta-objeto receptor analiza la cola de mensajes para decidir qué hacer con ellos.

De la misma forma que el objeto origen cede parte de su tiempo a su meta-espacio para que envíe el mensaje, también se cede parte del tiempo al meta-espacio del objeto receptor para que reciba el mensaje. Es decir, se cede tiempo para realizar la operación completa igual que si la ejecutase la máquina abstracta, directamente.

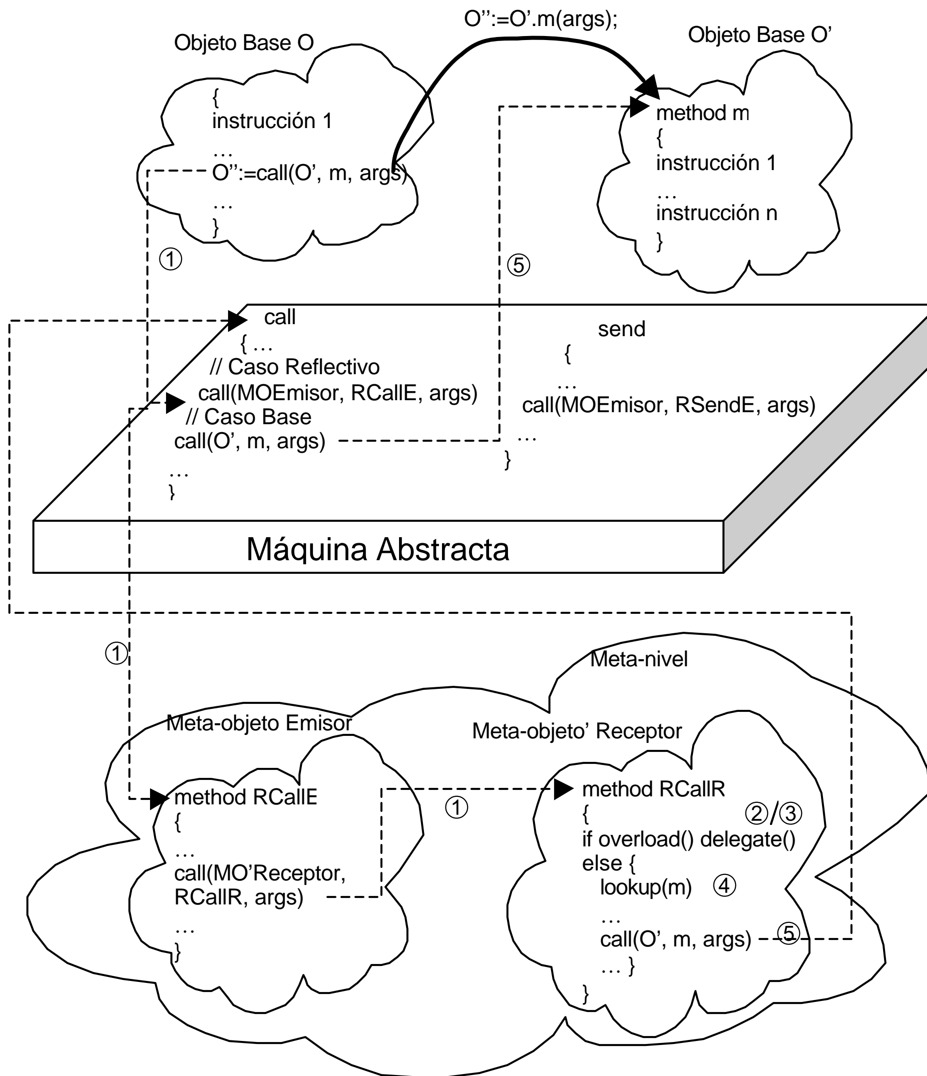
Comportamiento del meta-objeto

Una vez que el mensaje ha llegado al meta-objeto receptor, este objeto llevará a cabo la siguiente secuencia de acciones, destinadas a cumplir la ejecución de la instrucción de invocación.

Recepción de mensajes

1. Se invoca el método `RCallR` del meta objeto receptor. Normalmente, será el meta-objeto emisor el que lo haga.
2. El objeto receptor estudia el mensaje recibido: origen, método solicitado, parámetros, etc, con el fin de decidir la ejecución efectiva del método. Puede analizar la carga de trabajo del objeto y decidir delegarlo, retrasarlo o ponerlo en ejecución.
3. Si continúa su ejecución, busca el método a ejecutar.
4. Se ejecuta la instrucción de invocación de la máquina, que comienza la ejecución del método, sin invocar al planificador. Se produce así una transferencia de control del meta-nivel al nivel base. El meta-objeto se queda esperando la finalización de la

ejecución del método. Cuando termine, puede realizar más acciones antes de retornar el control al meta-objeto emisor del objeto origen.



① Transferencia del nivel base al meta-nivel. Ejecución síncrona del método RCallR en el meta-objeto Receptor para la transmisión del mensaje. Se suspende la ejecución del método origen y del método en el meta-objeto Emisor

② y ③ Análisis del mensaje y toma de decisión

④ Búsqueda del método adecuado para el mensaje recibido

⑤ Invocación efectiva del método mediante la ejecución de la instrucción call. El retorno síncrono irá reanudando la ejecución del método RCallR, RcallE y el método origen, retornando el resultado que se haya

Figura 9.7. Invocación síncrona de un método

El meta-objeto receptor en la invocación remota de métodos

De la misma forma que el meta-objeto emisor es modificado para satisfacer las necesidades de la invocación remota, también puede ser necesario modificar el comportamiento del meta-objeto receptor.

Por ejemplo, si el sistema de distribución de objetos dispone de una facilidad de replicación de objetos, es posible que el meta-objeto receptor redirija determinados mensajes a otras réplicas de su objeto base que se encuentren menos cargadas o que se encuentren en nodos menos cargados.

También relacionado con la replicación de objetos, se verá en el capítulo 13 cómo los meta-objetos receptor juegan un papel muy importante para mantener sincronizadas diferentes réplicas del mismo objeto.

9.5 Ventajas del uso de una arquitectura reflectiva para introducir la distribución en el sistema integral

Existe una serie de ventajas fundamentales derivadas de construir un sistema integral con una arquitectura reflectiva que se exponen a continuación [YW89, Smi90, KG96, CS98].

9.5.1 Mantenimiento de la uniformidad conceptual

La representación del entorno de ejecución de los objetos se realiza por medio de un conjunto de entidades, objetos a su vez. De esta forma, las abstracciones y mecanismos que proporcionan la funcionalidad del sistema operativo se proporcionan en forma de objetos (única abstracción en el sistema).

La relación de meta-objetos descritos (emisor, receptor, planificador, sincronizador, etc.) y los que se presentarán en los próximos capítulos son objetos como los de nivel usuario. El programador y el administrador del sistema operativo, programarán sus objetos utilizando en ambos casos el lenguaje proporcionado por la máquina abstracta, aunque con diferentes objetivos. El primero, con el fin de construir sus aplicaciones. El segundo, con el objetivo de adaptar el sistema operativo a un entorno concreto.

9.5.2 Extensibilidad y adaptabilidad

La utilización de la reflectividad establece una **arquitectura abierta**. Un sistema con una arquitectura reflectiva proporciona un medio explícito y modular para modificar dinámicamente la organización interna del sistema. Esto permite al programador cierto control sobre aspectos computacionales del sistema de forma modular y dinámica, permitiendo con ello implementar nuevas políticas sin cambiar el código de la aplicación.

Sería posible pensar en que en el sistema integral coexistiesen diferentes semánticas de invocación de objetos. Para ello, únicamente habría que crear diferentes parejas de meta-objetos emisor y receptor y asociar cada una de ellas a un conjunto de objetos que deseasen compartir la misma semántica de invocación.

9.5.3 Separación de asuntos o incumbencias (*concerns*)

Uno de los principios de diseño más importantes a la hora de diseñar sistemas operativos consiste en separar la política de los mecanismos [Cro96]. En este caso, la utilización de la reflectividad como medio de estructurar el sistema promueve la aplicación de este principio [KG96].

En la programación tradicional, los programas se mezclan y se ven complicados por el código que representa los algoritmos que los gestionan. Esto dificulta considerablemente entender, mantener, depurar y validar los programas.

Una arquitectura reflectiva proporciona interfaces a cada uno de los niveles de funcionalidad: una interfaz al nivel base y una interfaz al meta-nivel.

Esta separación del nivel base (algoritmo base) y el meta-nivel (nivel reflectivo) facilita la reutilización de políticas sin dañar el código base. El uso de un meta-espacio

separado permite al programador concentrarse en el dominio de la aplicación. La funcionalidad adicional se suministra mediante meta-componentes, que trabajarán con los objetos definidos por la aplicación. Es decir, se separa el aspecto puramente funcional de un objeto (dominio del problema) de otras incumbencias no funcionales (como distribución, persistencia, concurrencia, seguridad) que se implantan en el meta-nivel y pueden cambiarse y reutilizarse en otros objetos.

En el caso de la distribución, el programador escribe sus objetos sin preocuparse del número de nodos sobre los que se va a ejecutar el sistema integral ni si sus objetos van a residir todos juntos en el mismo nodo o van a tener la posibilidad de migrar. Las tareas relacionadas con la gestión de la distribución se realizarán en el meta-nivel y serán compartidas por un determinado conjunto de objetos del nivel base.

9.5.4 Favorece el diseño de un modelo de objetos activo

La implantación de un modelo de objetos activo se consigue fácilmente extendiendo la semántica tradicional del objeto, ampliamente extendida y conocida de los modelos de objetos pasivos, que definen los objetos como meros contenedores de datos y dotando a estos de un contenido semántico mucho mayor, que favorecerá el incremento del paralelismo.

Así, cada objeto deberá estar compuesto de datos + métodos + computación. Mientras que la parte pasiva del objeto implementa los aspectos funcionales del mismo, el comportamiento activo del objeto se define en el meta-nivel [GK98b].

Como ya se ha comentado anteriormente (ver capítulo 3) la utilización de un modelo de objetos activo facilita las tareas de la distribución, principalmente la migración de objetos. Dado que la arquitectura reflectiva favorece la implantación de un modelo de objetos activo y éste facilita las tareas relacionadas con la distribución, de manera indirecta se obtiene un beneficio extra de la reflectividad.

9.5.5 Configurabilidad

El soporte de un meta-nivel facilita una arquitectura abierta [Kic96] en la que se pueden implementar nuevas políticas sin cambiar el código de la aplicación.

Esta propiedad se consigue, no sólo a nivel de los desarrolladores de aplicaciones, que se benefician de la meta-programación para desarrollar aplicaciones configurables, sino también a nivel de los usuarios, que pueden reemplazar meta-componentes para personalizar las aplicaciones de acuerdo a sus necesidades específicas o a las características del entorno en el que se van a ejecutar.

9.6 Resumen

Con el fin de que el sistema operativo pueda colaborar en la definición del funcionamiento de la máquina se dota a esta de una arquitectura reflectiva. En este tipo de arquitecturas los objetos del sistema base pueden acceder y manipular su meta-sistema (objetos de la máquina) que les dan soporte. Para ello se utilizan dos operaciones: exposición (acceso al meta-sistema desde el nivel base) y reflexión (efecto de la exposición en el meta-sistema).

Para crear una perspectiva uniforme del sistema se propone describir los objetos de la máquina con el mismo modelo y dentro del mismo espacio de los objetos de usuario, unificando las operaciones de exposición y reflexión con la invocación de métodos. La programación de los objetos normales y los de la máquina (meta-programación) se

unifica. En lugar de un modelo con mucho detalle, que podría complicar la comprensión del sistema se utilizarán como meta-objetos las áreas constituyentes de la máquina, no permitiendo que su interfaz deje cambiar las propiedades fundamentales del sistema (para mantener la consistencia).

Esto aporta una gran flexibilidad, puesto que permite la extensión dinámica del sistema por medio de objetos en el espacio de usuario. La reflectividad hace que los propios objetos de la máquina se definan en el espacio de usuario, permitiendo aplicar todas las ventajas del espacio de usuario a los propios objetos de la máquina: eliminación, adición y modificación dinámica de los mismos sin necesidad de detener el sistema y recompilar la implementación primitiva del sistema.

CAPÍTULO 10 CUESTIONES INICIALES DE DISEÑO DE AGRA

10.1 Introducción

Anteriormente, se definió el sistema integral orientado a objetos (SIOO) como una combinación de la máquina abstracta y el sistema operativo. El sistema integral tendrá una **arquitectura distribuida** (SIOOD) cuando exista una serie de máquinas abstractas conectadas en red y el **sistema operativo extienda la máquina** abstracta implementando las características de distribución. De aquí en adelante se utilizará el término **nodo** para hacer referencia a una máquina abstracta que forma parte del sistema integral distribuido.

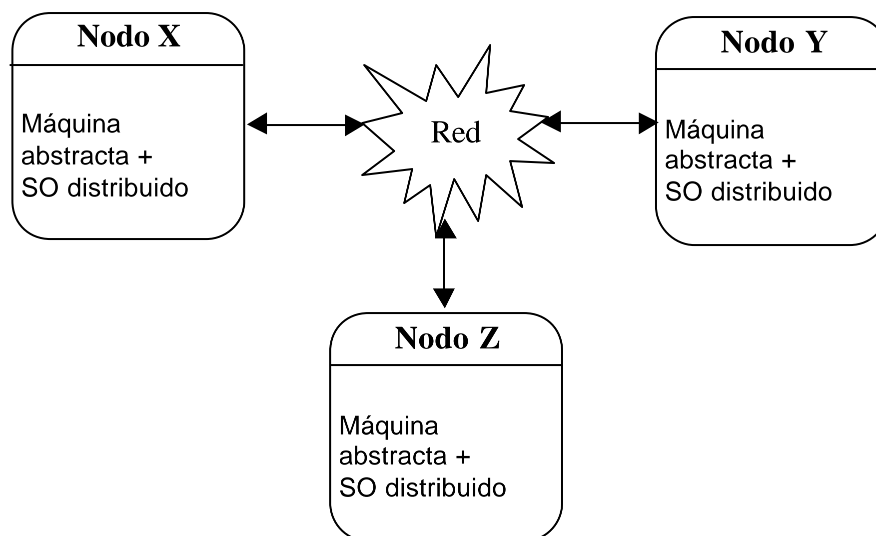


Figura 10.1. Arquitectura distribuida del sistema integral.

Se pretende, por tanto, extender las capacidades del sistema integral para aumentarlo con los beneficios ya descritos de la distribución, en general, y los de la orientación a objetos, en particular.

Como resultado de la introducción de la distribución como característica del sistema operativo, se conseguirá que los objetos del sistema integral puedan **interaccionar sin preocuparse de su ubicación** (transparencia), es decir, el usuario programador será ajeno (aunque podrá ser consciente, si lo desea) a la existencia de la distribución, pudiendo concentrarse en la resolución del problema (separación de incumbencias).

Adicionalmente, se desea que los mecanismos y políticas relacionados con la distribución puedan ser **adaptados** a las necesidades concretas del entorno en el que va a funcionar el sistema integral: entorno centralizado, entorno distribuido en red LAN, entorno distribuido en red WAN, etc.

10.2 Introducción de la distribución en el sistema integral

Como ya se comentó en el capítulo 6, es necesario dotar a las máquinas abstractas de una **arquitectura reflectiva** para lograr la extensibilidad y adaptabilidad de las mismas sin romper la uniformidad en la utilización de la orientación a objetos.

Todos aquellos mecanismos que modifiquen el comportamiento por defecto de las máquinas abstractas serán introducidos en el sistema operativo. Las máquinas abstractas cederán el control al sistema operativo para aquellas operaciones que tengan redefinido su comportamiento en él. El **sistema operativo queda definido por un conjunto de objetos especiales**, denominados meta-objetos, cuyo comportamiento se especifica en el propio lenguaje de la máquina abstracta, algunos de los cuales ya se describieron en el capítulo anterior.

Adicionalmente, las máquinas abstractas proporcionarán una visión de sus elementos estructurales en forma de objetos, de tal forma que puedan ser utilizados por el procedimiento usual de invocación de sus métodos.

La distribución es una característica de algunos sistemas operativos. Su introducción en el sistema integral lo convierte en un sistema integral y distribuido de objetos (SIOOD) que le permite beneficiarse de todas las ventajas de la orientación a objetos y de la distribución de los mismos.

La **introducción de la distribución** en el sistema integral, con la deseada transparencia para los objetos de usuario, se realizará, como característica de sistema operativo que es, con un conjunto de **objetos que modificarán el comportamiento por defecto** de la máquina abstracta. Será necesario adaptar el comportamiento mostrado para los meta-objetos básicos vistos en el capítulo anterior a las necesidades de la distribución de objetos, así como definir nuevos meta-objetos que completen las características deseadas del sistema de distribución.

El **diseño de AGRA**, el subsistema de distribución del sistema integral, se corresponde, pues, con el diseño del conjunto de meta-objetos que residen en el meta-nivel, su relación e interacción con las máquinas abstractas, la definición del modelo de objetos soportado, su influencia en la estructura interna de los objetos, la definición de los mecanismos de comunicación entre diferentes máquinas abstractas y la posibilidad de definir diferentes políticas para cada uno de los mecanismos existentes (separación política/mecanismo) [ATD+2000].

AGRA no se presenta como un sistema de distribución de objetos con todas las características presentadas en los capítulos 2 y 3 Únicamente extiende el soporte de objetos básico proporcionado por una máquina abstracta (creación y eliminación de objetos, invocación de métodos, etc.) para un entorno distribuido de máquinas abstractas, ofreciendo **la migración de objetos y la invocación remota de los mismos como características exclusivas de la distribución** y convirtiendo el objeto, de cualquier granularidad, en la **unidad de distribución**. Servicios como transacciones, replicación, servicios de nombrado, etc., íntimamente relacionados con la distribución, no se consideran como integrantes del núcleo de distribución del sistema integral, aunque serán igualmente tratados en capítulos posteriores.

10.3 Cuestiones de diseño impuestas por el sistema integral

La cuestión fundamental impuesta por el sistema integral para el diseño de AGRA es la utilización del **objeto como único nivel de abstracción**.

10.3.1 Uniformidad en torno a la orientación a objetos

La máquina abstracta proporciona el soporte básico para los objetos, y, con este, el modelo único de objetos del sistema. Además, la máquina dispone de una **arquitectura reflectiva**, que permite que los propios objetos constituyentes de la máquina (áreas) y la propia estructura de los objetos (atributos, métodos, clases, instrucciones, etc.) puedan usarse dentro del sistema como cualquier otro objeto. Por su parte, el sistema operativo, está construido también como un conjunto de objetos.

La combinación de la máquina abstracta con el sistema operativo produce un espacio de objetos en el que residen los objetos. No existe una división entre los objetos del sistema y los del usuario. Todos están al mismo nivel, independientemente de que se puedan considerar objetos de aplicaciones normales de usuario u objetos que proporcionen funcionalidad del sistema.

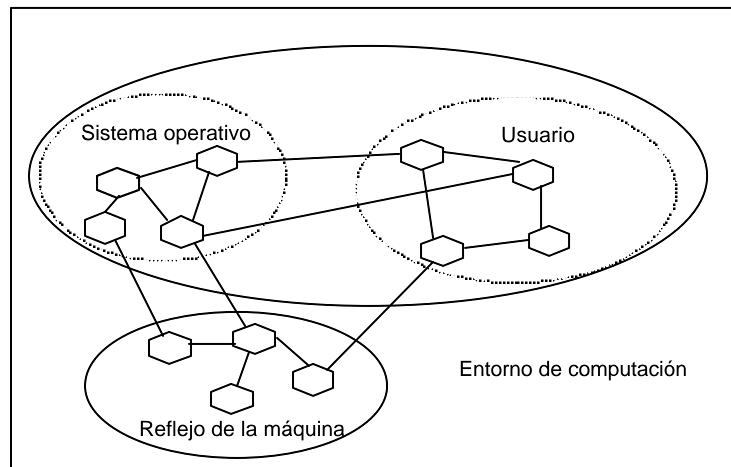


Figura 10.2. Espacio de objetos homogéneos en un nodo.

Con la introducción de la distribución, el espacio de objetos único se obtiene como la unión de los espacios de objetos proporcionados en cada uno de los nodos del sistema integral (ver siguiente figura).

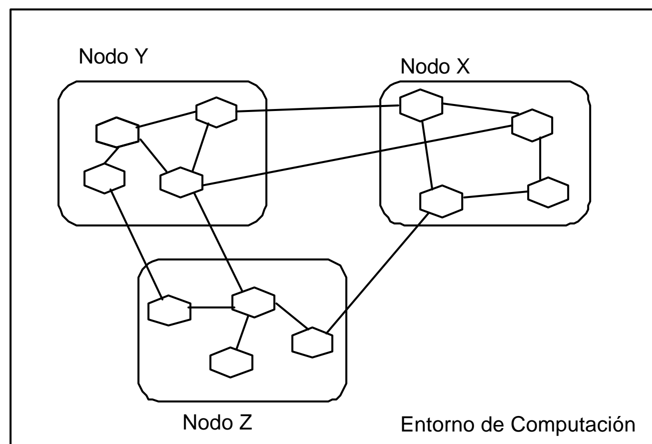


Figura 10.3. Espacio único de objetos homogéneos en el sistema integral distribuido.

10.3.2 Identidad de los objetos

La identificación de los objetos es necesaria en cualquier sistema de gestión de objetos e imprescindible si se habla de un sistema distribuido de objetos. El

identificador de un objeto será la información mínima necesaria para las operaciones de búsqueda de objetos y de comparación (de igualdad) entre diferentes referencias a objetos.

Los objetos dispondrán de un **identificador único dentro del sistema**. Este identificador será válido dentro de todo el sistema integral y será el único medio por el que se pueda acceder a un objeto. Además, el identificador será **independiente de la ubicación del objeto**, para hacer transparente la ubicación de un objeto al resto del sistema.

10.4 Características de los objetos

En el capítulo 3 se describen diferentes aspectos a tener en cuenta a la hora de definir la estructura que van a presentar los objetos. A continuación se discuten las distintas posibilidades y se describen las decisiones adoptadas, algunas de las cuales ya vienen determinadas por el modelo de objetos del sistema integral y otras por el diseño del propio sistema de distribución de objetos.

10.4.1 Objetos activos autocontenidos

A la hora de elegir el modelo de computación de un sistema de gestión de objetos, surgen fundamentalmente los siguientes: modelo de objetos pasivo y modelo de objetos activo (ver capítulo 3).

En el capitulo 5 se planteó como requisito del mismo la adopción de una uniformidad conceptual en torno a la orientación a objetos, donde la única abstracción es el objeto y toda su semántica se encuentra encapsulada en él. Adicionalmente, todos los objetos deben tener la misma categoría, es decir, los propios objetos que den soporte al sistema no deben ser diferentes del resto de los objetos (los de los usuarios). En este sentido, el modelo de objetos a adoptar por el sistema integral es el modelo activo. Más adelante, dentro de este apartado, se mostrará incluso que el modelo activo de objetos favorece la distribución.

En el sistema integral, los **objetos se convierten en la unidad de procesamiento** [Taj97], capaces de ejecutar sus métodos ante solicitudes por parte de otros objetos. La computación o procesamiento se representa como una secuencia de mensajes entre objetos. Dado que un objeto es una entidad autocontenida y dotada de un protocolo de comunicación unificado, la descomposición de un sistema en un conjunto de objetos resulta muy flexible.

Es más, en un sistema OO donde todo está encapsulado en forma de objetos (entidades autocontenidas) y la única forma de acceder a un objeto es solicitando un servicio que él mismo debe proporcionar y gestionar (invocación a métodos), la propia semántica del objeto protege a los mismos de accesos ilegítimos, dando soporte a una interacción ordenada entre objetos [YTT88a].

Con esto, gran parte del significado del proceso o tarea pierde su sentido ya que esta entidad servía como elemento de protección del espacio de direcciones frente a otros procesos que podían, incluso intencionadamente, intentar realizar un acceso ilegítimo.

La definición de entidad autocontenida no sólo es suficiente para asegurar la no intromisión externa sino que, a la vez, garantiza la propiedad de encapsulación.

10.4.1.1 Relación con la distribución

El modelo de computación utilizado en el sistema y las tareas propias de la distribución de objetos está íntimamente relacionadas. La forma en que se procesa una invocación remota y la posibilidad de mover objetos entre distintos nodos del sistema integral está muy influenciado por el modelo de computación.

Modelo de objetos pasivo

En un modelo de objetos pasivo existe una entidad proceso (o hilo) que va ejecutando diferentes métodos en diferentes objetos, siguiendo el conjunto de llamadas a método encadenadas que se va encontrando. El estado de la computación se almacena en el proceso, en una estructura de datos de tipo pila, con una operación de apilado por cada nueva llamada a método y una operación de desapilado que se realiza cuando el método que ocupa la cima de la pila termina. En el caso de una invocación remota (ver figura siguiente), se crea un **proceso trabajador** (*worker*) en el nodo remoto (1.1) y (1.2), el cual trabaja a instancias del proceso cliente. El proceso trabajador resuelve la invocación de manera local (3) y (3.1) y devuelve el resultado al proceso cliente (3.2).

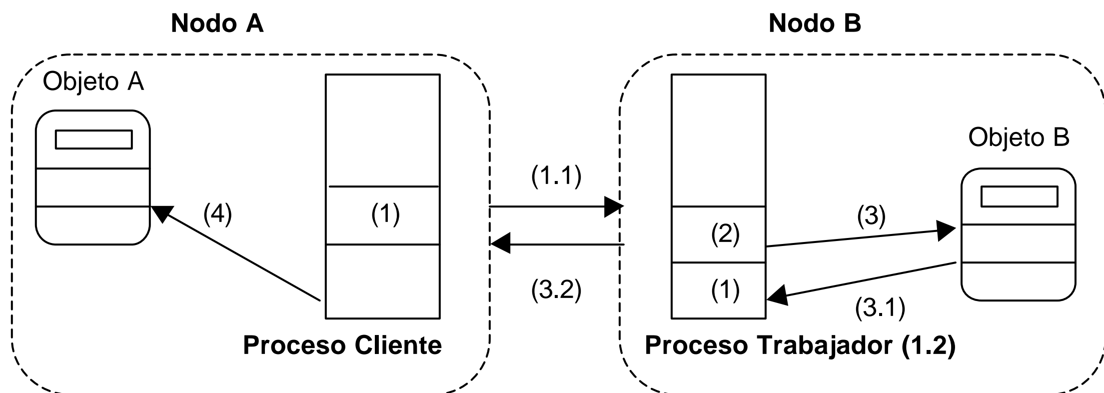


Figura 10.4. Invocación remota en el modelo de objetos pasivo.

El proceso trabajador, a su vez, dispone de una pila donde almacena el estado de su computación. Este proceso, ejecutando el método solicitado sobre el objeto remoto, podrá realizar invocaciones a métodos de otros objetos, posiblemente remotos. Tendremos, en general, que una computación iniciada por un proceso estará distribuida por un conjunto de procesos (el original más un conjunto de procesos trabajadores) que contendrán un estado parcial de la computación relativa al objeto local con el que interactúan.

La separación entre un objeto y su computación **dificulta la migración**. Por un lado, el objeto contiene los datos que conforman su estado interno. Por otro lado, y dentro de los procesos, se encuentra toda aquella información relativa a su computación y a la de otros objetos. Si se desea migrar un objeto, es necesario migrar la información sobre su computación, dado que la computación afecta generalmente al estado interno del objeto. Pero la migración de un proceso mueve también la información sobre la computación de otros objetos, lo cual obligaría a mover esos objetos también. En otro caso, sería necesario poder extraer del proceso la información sobre la computación del objeto para poder migrarla con el objeto. Como, en general, un objeto puede estar atendiendo varias invocaciones simultáneamente, sería necesario extraer la computación de varios procesos. Finalmente, sería necesario crear procesos trabajador para realizar la

computación del objeto migrado en el nodo destino y relacionar dichos procesos con los procesos que se modificaron en el nodo original.

Aparte de la dificultad mencionada para la migración de objetos, el modelo de objetos pasivo **rompe la uniformidad** de los objetos del sistema, al coexistir un tipo de objetos meros contenedores de datos y otro tipo de objetos radicalmente distintos responsables de la computación que se realiza en los primeros.

10.4.1.2 Modelo de objetos activo

En un modelo de objetos activo no se distinguen objetos proceso del resto de los objetos. Un objeto cliente invoca un método de un objeto servidor (cliente y servidor sólo indican quién invoca y quién es invocado) que acepta la invocación y ejecuta el método correspondiente, valiéndose para ello de las actividades internas que el sistema de soporte le proporciona. La invocación propiamente dicha consistirá en un mensaje que contendrá los parámetros de la invocación y que se hará llegar al objeto servidor desde el objeto cliente. El **estado de la computación estará encapsulado** en cada objeto, es decir, cada objeto almacenará el estado de la computación de las invocaciones a sus métodos.

Invocación remota

En el caso de una invocación remota, los objetos cliente y servidor se van a comportar de la misma manera que en el caso de la invocación local. La única diferencia entre una invocación local y una remota será la forma en que se haga llegar el mensaje al objeto servidor: el mecanismo de paso de mensajes deberá ser capaz de cruzar fronteras entre diferentes máquinas/nodos.

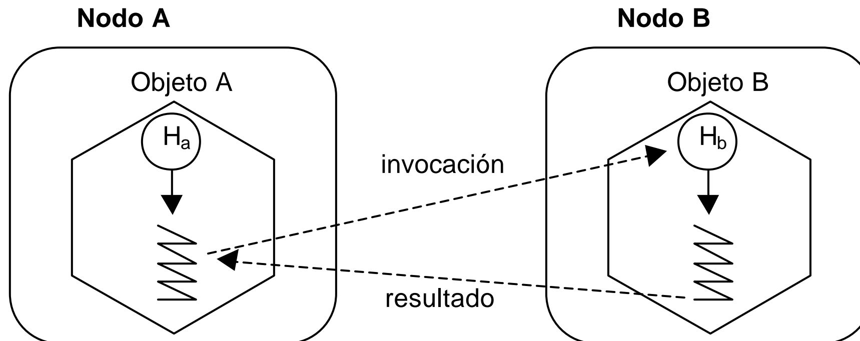


Figura 10.5. Invocación remota en el modelo de objetos activo.

Migración de objetos

La migración de un objeto activo consiste únicamente en la **transmisión del estado encapsulado del objeto** del nodo origen de la migración al nodo destino. Dicho estado encapsulado contiene tanto el estado propiamente dicho del objeto como el estado de su computación interna. Dado que todo objeto activo almacena la identidad de los objetos que le han invocado y cuya computación está en curso, la finalización de una invocación únicamente tiene que hacer llegar un mensaje con el resultado a dicho objeto. La llegada de dicho mensaje al objeto que realizó la invocación le permitirá a su vez continuar con su propia computación.

10.4.1.3 Modelo de objetos para el sistema integral

Los objetos activos (autocontenidos) están, por tanto, más dispuestos para ser distribuidos de una manera sencilla. Utilizando un razonamiento similar, la persistencia de objetos también se consigue de una manera más simple utilizando este tipo de objetos [**REFERENCIA A ALGUIEN**]. El **modelo de objetos activo será, entonces, el escogido** para el sistema integral.

10.4.2 Granularidad de los objetos

El tamaño, sobrecarga y cantidad de procesamiento realizado por un objeto caracterizan su **granularidad**. En un sistema integral, donde la única abstracción es el objeto, no se puede hablar de una granularidad concreta de los objetos, ya que se puede encontrar una pequeña cantidad de objetos de grano grueso, una considerable cantidad de objetos de grado medio, y una gran cantidad de objetos de grano fino. El sistema integral (máquina abstracta y sistema operativo) tratará de manera uniforme los objetos, independientemente de su grano, siguiendo las líneas maestras que definen la arquitectura del sistema.

En general, los distintos granos de objetos juegan papeles diferenciados en el sistema. Los objetos de pequeño tamaño (entero, lógico, etc.) juegan el papel de agregados de objetos de mayor tamaño, y sólo son conocidos y manipulados por los métodos de estos, teniendo un carácter más bien privado. Los objetos públicos (compartidos), que proporcionan servicios a cierta cantidad de otros objetos, serán en general objetos de grano medio o grueso. Estos objetos se ofrecerán al resto de los objetos del sistema a través de algún servicio de nombrado (construido con un conjunto de objetos de nivel usuario), de manera que exista un medio a través del cual todo objeto puede obtener una referencia a ellos.

10.4.2.1 Relación con la distribución

Existe una relación muy fuerte entre la granularidad de los objetos del sistema y otros aspectos de diseño del mismo.

El hecho de que exista un número muy elevado de objetos en el sistema complica las tareas básicas de gestión de objetos. A la hora de asignar identificadores diferentes a los objetos del sistema, es preciso garantizar que los identificadores sean distintos para todos los objetos, incluso si la asignación va a tener lugar en diferentes nodos del sistema integral.

A la hora de mover objetos, puede no ser rentable (eficiente) mover individualmente objetos de pequeño tamaño ni objetos de gran tamaño. Para los objetos de pequeño tamaño se suelen utilizar **políticas de agrupación**, que construyen, implícita o explícitamente, grupos de objetos relacionados con el objetivo de optimizar las operaciones de migración. Las políticas de migración de objetos tendrá que tomar decisiones que tengan en cuenta el tamaño de los objetos y las posibilidades de agrupación mencionadas.

La replicación de objetos puede mejorar la eficiencia del sistema. Los objetos susceptibles de ser replicados serán aquellos que juegan un papel público y que se corresponden con objetos servidores de recursos, de un tamaño mediano/grande, que sirven a un conjunto moderadamente numeroso y distribuido de objetos clientes.

Es necesario **coordinar las políticas de migración y replicación** (si existe), de modo que se evalúe en cada momento si un objeto da mejor servicio siendo movido a otra máquina virtual o siendo replicado en dicha máquina virtual.

10.4.2.2 Granularidad de los objetos distribuidos

El sistema integral orientado a objetos proporciona soporte para objetos, con independencia de su granularidad. La característica de la distribución afectará a todos por igual, de manera que cualquier objeto puede migrar y ser invocado remotamente. La existencia de mecanismos de replicación y políticas de agrupación permite obtener el máximo rendimiento del hecho de que **todos los objetos son iguales a la vista de los mecanismos de distribución**.

10.4.3 Objetos compartidos

En un principio, y dado que todos los objetos y sus referencias van a estar almacenados en áreas de la máquina abstracta, se pueden suponer todos los objetos compartidos (públicos).

Con el fin de que los objetos puedan interactuar, será necesario que al menos uno de ellos invoque métodos del otro o que ambos invoquen métodos de un tercero (compartido). Evidentemente, es necesario que los objetos conozcan la identidad de los objetos que desean invocar, es decir, que posean alguna referencia a esos objetos. Se puede pensar en que aquellos objetos cuya identidad (existencia) es desconocida para otros son objetos privados (no compartidos). Finalmente, y a pesar de que un objeto conozca la existencia de otro, es posible que no pueda hacer uso de sus servicios por no estar autorizado para ello (no tiene los permisos adecuados), por lo que para ese objeto en concreto se diluye la idea de compartición.

En general, se puede decir que existe un conjunto de objetos compartidos diseñados con el objetivo de proporcionar determinados servicios (por ejemplo, un servicio de impresión) al resto, y que existe después otro grupo de objetos auxiliares de ese primer grupo que se utilizan como objetos privados. El grado de privacidad/publicidad de un objeto vendrá determinado por la dificultad/facilidad de obtener referencias al mismo para solicitar sus servicios.

10.4.3.1 Objetos, privados y compartidos, distribuidos

El modelo cliente/servidor se apoya en la existencia de ciertas entidades (servidores) que prestan sus servicios a otras entidades. En la mayor parte de los casos, los servidores existen para posibilitar la utilización ordenada de recursos compartidos. En el sistema integral, **todo recurso compartido (o no) toma la forma de un objeto**.

Los objetos servidores, en el sentido del párrafo anterior, se registran a su vez en servidores concretos, que hacen posible a otros objetos conocer su existencia y, lo que es más importante, su identidad, que les permitirá, en definitiva, solicitar sus servicios.

Existe también un conjunto, si cabe mayor, de objetos compartidos que no se publican para poder ser utilizados. Son aquellos objetos que se pasan como parámetros en las invocaciones a métodos de otros objetos, y que, inmediatamente, son conocidos por el objeto receptor de la invocación. La publicidad de este tipo de objetos es transitoria y limitada, dado que termina tan pronto como van terminando los métodos invocados y sólo son conocidos por los objetos en los que residen dichos métodos.

En el sistema integral coexistirán, por tanto objetos privados y compartidos. La facilidad de distribución permitirá extender el uso de los objetos (ya sea un uso privado o compartido) en toda la extensión del sistema integral, y de una manera transparente. El acceso a los objetos privados se realizará directamente a través de las referencias, como ya se ha explicado. Para el acceso a los objetos compartidos, será necesaria la **creación de un servicio de nombrado distribuido**, en el que se registren todos ellos,

con el fin de que el resto de objetos tenga la posibilidad de obtener referencias que les permitan solicitar sus servicios.

10.4.4 Objetos de una copia

En un entorno de máquinas abstractas distribuidas, donde la computación se basa en la interacción entre objetos localizados (supuestamente) de manera dispersa en el sistema, se lograría un mejor rendimiento global si todas las invocaciones entre objetos se pudieran resolver dentro de una misma máquina abstracta. Aunque con la utilización de la característica de movilidad de los objetos se podría conseguir en buena medida que esto fuera así, nunca se llegaría a conseguir del todo porque:

- El objeto invocado está sirviendo simultáneamente múltiples peticiones de objetos localizados en máquinas abstractas distintas, o
- El coste de mover frecuentemente el objeto invocado para colocarlo en la misma máquina abstracta desde la que es invocado es muy alto.

Puede ser conveniente, entonces, que aquellos objetos que se encuentren en alguna de esas situaciones se "comporten" de una manera ligeramente distinta que la especificada para el resto de los objetos, siempre con el objetivo de mejorar el rendimiento y la fiabilidad del sistema. La solución pasa por disponer de tantas réplicas del objeto como sean necesarias para conseguir que todas (o la mayor parte) de las invocaciones al mismo se resolvieran de manera local, o bien que la indisponibilidad de una máquina abstracta (por una caída repentina) no provocase la imposibilidad de invocar al objeto replicado. En el caso de que el objeto sea de sólo lectura (objeto inmutable, ninguna invocación a método modifica su estado), parece que no hay problema para la replicación. Los problemas surgen cuando el estado del objeto puede modificarse debido a la invocación de un método.

10.4.4.1 Replicación de objetos como característica avanzada de la distribución

En cualquier caso, el modelo de objetos que proporciona la máquina abstracta **no proporciona el concepto de objeto replicado**, ya que se ha definido con el objetivo de ser lo más simple posible. Dado que la replicación de objetos no se utiliza, en general, para todos los objetos del sistema, sino más bien para objetos que juegan el papel de servidores públicos, la **replicación será una abstracción ofrecida al nivel del sistema operativo**. Como beneficio importante de la replicación de objetos aparece la mejora de la escalabilidad del sistema, al poder evitarse la existencia de componentes (objetos servidores públicos) centralizados.

En el capítulo 13 se describe la forma en que se puede introducir la replicación de objetos en el sistema integral.

10.4.5 Objetos persistentes

En los sistemas operativos tradicionales se utilizan los ficheros para conseguir que determinada información se mantenga entre diferentes periodos de conexión de la máquina. Dichos sistemas operativos proporcionan conjuntos de operaciones diferentes para tratar la información almacenada en la memoria principal y la información almacenada en los ficheros. Esto obliga a que el programador de aplicaciones tenga que ocuparse de la transferencia de información entre la memoria principal y el almacenamiento secundario y viceversa. Este problema es un ejemplo típico de la denominada **desadaptación de impedancias** [ABB+93, Álv98].

En un sistema operativo orientado a objetos, la necesidad de que la información pueda mantenerse entre diferentes periodos de conexión sigue existiendo. Un **sistema de persistencia de objetos** permitiría que todos los objetos (tanto los almacenados en memoria principal como en memoria secundaria) se tratasen de la misma forma, con el mismo conjunto de operaciones (invocación a métodos). De la misma manera que ocurre con la distribución, es imprescindible que el sistema de persistencia sea transparente para que el programador no tenga que preocuparse de la situación instantánea de un objeto.

10.4.5.1 Objetos persistentes distribuidos

La coexistencia de la distribución y la persistencia permite crear un **espacio de objetos virtualmente infinito en el espacio (distribución) y en el tiempo (persistencia)**.

Con el fin de dotar de persistencia a los objetos del sistema, la máquina abstracta colaborará de manera reflectiva con el sistema operativo, de una manera similar a como lo hace para la introducción de la distribución.

En el capítulo 14 se describe de manera resumida el diseño del mecanismo de persistencia para el sistema integral [Álv98] y de qué manera se relaciona con el sistema de distribución.

10.4.6 Objetos móviles

En diferentes trabajos sobre migración [AF89, Che88, MPP86, PM83] se resalta la migración de entidades de gran tamaño, donde un proceso o, en general, un espacio de direcciones, se mueve en una operación. La entidad más pequeña con capacidad de movimiento es el proceso, y no es posible mover entidades que residen en su espacio de direcciones, como por ejemplo, un registro. Se habla en estos casos de una **migración de grano grueso**.

Otros trabajos [Jul88], en cambio, proporcionan una **migración de grano fino**, donde el objeto, independientemente de su tamaño, se convierte en la entidad más pequeña con capacidad de movimiento.

En el sistema integral, donde la única abstracción que existe es el objeto, inmediatamente se deduce que el **objeto se va a convertir en la unidad elemental de migración**. Dado que los objetos del sistema integral encapsulan datos y computación (objetos activos), una operación de migración de un objeto comprenderá los mecanismos tradicionales de movimiento de datos y migración de procesos (para ser más precisos, hilos).

La idea básica de la migración de objetos es que estos se puedan mover durante su actividad normal, de tal forma que puedan continuar con ella en otra máquina abstracta. La operación de migración puede ser iniciada sin conocimiento del propio objeto, lo que implica que la operación se tiene que hacer de modo transparente.

La migración de objetos es muy útil en un entorno distribuido de máquinas abstractas, y tiene dos objetivos básicos:

- **Equilibrar la carga** entre las diferentes máquinas abstractas.
- **Minimizar el número de invocaciones remotas** a métodos.

Tal y como se hace en las arquitecturas y sistemas operativos tradicionales, es beneficioso para el rendimiento global del sistema conseguir que la carga de

computación del mismo en todo momento esté lo mejor repartida posible entre las distintas unidades de cómputo, de tal manera que la computación global pueda realizarse de una manera más eficiente con la utilización de máquinas ociosas o con escasa carga.

Por otro lado, si las invocaciones a métodos se pueden resolver de manera local (es decir, en la misma máquina abstracta), se va a obtener un beneficio añadido para el rendimiento global, al evitarse los costes asociados a la comunicación entre distintos nodos del sistema integral (en algunos casos, la diferencia entre la comunicación remota y su resolución local puede ser de tres órdenes de magnitud [Jul88]). Para ello es necesario que los objetos invocador e invocado se encuentren, en el mayor número posible de casos, ubicados en la misma máquina abstracta.

Sea cual sea la razón por la que se desee mover un objeto, hay que tomar en consideración un conjunto de cuestiones: cúando mover, qué mover, adónde mover y cómo mover.

- **Cuándo mover**: básicamente, un objeto se moverá cuando el sistema operativo así lo decida, aunque los objetos de usuario podrán solicitar al sistema operativo la migración de objetos, si desean llevar un control más preciso de su ubicación. El sistema operativo tendrá que concretar en qué instantes se van a producir operaciones de migración, como por ejemplo: de manera periódica, cuando se produzca determinado evento, etc.
- **Qué mover**: se trata de decidir cuál será la unidad real de migración. La unidad mínima de migración es el objeto (de cualquier granularidad), aunque en general no será rentable mover objetos individuales (más aún, si son de pequeño tamaño), sino mover conjuntos de objetos relacionados, con el fin de que los objetivos mencionados no se desvirtúen. Además, puede no ser rentable realizar muchos movimientos de objetos de pequeño tamaño (granularidad) [BHD+98].
- **Adónde mover**: la elección de los nodos de destino de los objetos a mover viene prácticamente predeterminado por el hecho de haber sido elegidos para el movimiento. En el caso de que la decisión de migrar un objeto se haya tomado por tener una importante interacción con objetos remotos, la elección del nodo destino parece sencilla, aunque es posible que el grado de interacción de ese objeto con objetos remotos esté repartido de una forma parecida con dos o más nodos del sistema integral. En el caso de que se desee equilibrar la carga de los nodos del sistema integral, la solución no parece tan inmediata, dado que puede haber múltiples formas de mover objetos para conseguirlo.
- **Cómo mover**: la operación que realiza definitivamente el movimiento del objeto es crítica (en tiempo y forma) e involucra a su vez un subconjunto de operaciones que se tienen que realizar con especial cuidado para que la operación en su conjunto tenga éxito.

Las tres primeras cuestiones son, fundamentalmente, procesos de toma de decisión, es decir, se deben concretar en políticas. Una vez tomadas todas y cada una de las decisiones correspondientes a los tres puntos anteriores, queda llevar a cabo la operación de migración propiamente dicha, es decir, realizar el cómo. El cómo es realmente la concreción de la característica de migración de los objetos, es decir, es el mecanismo por el cual todo aquello que se haya decidido que ha de migrar, migra desde su ubicación de origen a su ubicación de destino.

En el capítulo 12 se describe en detalle cómo se concreta en el sistema integral este conjunto de aspectos imprescindibles para la migración de objetos.

10.4.7 Enlace dinámico

En el sistema integral, todos los métodos de los objetos son virtuales (en el sentido de C++), es decir, se utiliza únicamente el mecanismo de enlace dinámico, siguiendo la línea de una arquitectura OO más pura. El uso de enlace estático restringe en exceso la extensibilidad del código, perdiéndose una de las ventajas de la OO. La posible sobrecarga de ejecución de los métodos virtuales se puede compensar con una implementación interna eficiente.

10.4.7.1 Relación con la distribución

El enlace dinámico se presenta como imprescindible para posibilitar la distribución y movilidad de los objetos. La naturaleza distribuida del sistema integral impide conocer, con total certeza, la ubicación de los objetos en tiempo de ejecución. Este hecho es extensible a la propia definición de los objetos, es decir, sus clases. Por tanto, no queda más remedio que resolver en tiempo de ejecución las referencias a métodos que hagan los objetos, teniendo que llegarse, en muchos casos, a realizar búsquedas en diferentes nodos del sistema integral.

10.5 Representación interna de los objetos

A continuación se describe la forma en que las máquinas abstractas dan soporte a los objetos, es decir, **de qué manera se disponen los objetos en el área de instancias** de la máquina y las relaciones que mantienen con entidades contenidas en otras áreas. En general, la estructura mostrada es válida para los objetos y los meta-objetos, aunque con un pequeño matiz que se comentará.

Identificador de objeto
Referencia a objeto (instancia) del que es subobjeto (subinstancia)
Array de referencias a objetos agregados
Array de referencias a objetos asociados
Referencia a meta-espacio
Referencia a clase
Referencia a clase sombra

Figura 10.6. Estructura interna de un objeto.

El **identificador de objeto** se corresponde con el identificador único y global que distingue a todo objeto del resto dentro del sistema integral.

Todo objeto agregado mantiene una referencia que apunta al **objeto del que es agregado**. En el caso de que el objeto no sea agregado de otro, esta referencia estará vacía.

Un objeto puede contener una serie de **atributos de tipo agregado**, que se crean y destruyen cuando se crea y destruye el propio objeto, y que pueden estar definidos en la clase del objeto o en alguna de sus superclases.

Adicionalmente, un objeto puede contener una serie de **atributos de tipo asociación**, que permiten mantener relaciones de asociación entre el objeto y otros objetos. Los objetos asociados se crean y destruyen con independencia de la creación y destrucción del propio objeto.

La **referencia al meta-espacio** permite obtener el conjunto de meta-objetos que, para este objeto en concreto, sustituyen a la máquina abstracta en determinadas tareas. Entre dichos meta-objetos se encuentran los responsables de la distribución, cuyo comportamiento se describirá principalmente en los capítulos 11 y 12. Para los meta-objetos la referencia al meta-espacio siempre será nula, dado que se adoptó una torre reflectiva de dos niveles (ver capítulo 9), y por tanto los meta-objetos no disponen de un meta-nivel.

Todo objeto pertenece a una **clase**, definida en algún nodo del sistema integral, por lo que es necesario la referencia a la clase.

Finalmente, se introduce la referencia a la **clase sombra**. La clase sombra es imprescindible para la gestión distribuida de objetos en tiempo de ejecución y su objetivo es mantener copias locales de toda aquella información necesaria y que no está disponible de manera local. Dicha información incluye la descripción de los atributos del objeto, una copia de todos aquellos métodos para los que existen hilos activos en el objeto e información para la comprobación de permisos de invocación de dichos métodos, que permite una comprobación de la protección más eficiente. En el apartado siguiente se justifica de una manera más amplia la necesidad de la clase sombra. En los capítulos siguientes se verán, de manera detallada, las actividades en las que juega su papel la clase sombra.

Todas las referencias anteriores tienen carácter global, es decir, pueden apuntar a cualquier nodo del sistema integral, excepto las que apuntan a los meta-objetos y la clase sombra, que siempre apuntarán a información local. Por tanto, si deseamos migrar un objeto, será necesario migrar a su vez toda la información de los meta-objetos y de la clase sombra que vaya a utilizarse en el nodo destino.

La figura siguiente ilustra como se representan internamente los objetos. En ella aparece un objeto, con identificador ID0987, que tiene dos agregados (ID3552 e ID7654) y un objeto asociado. Se puede comprobar como los objetos agregados guardan la relación con el objeto del que lo son. La clase a la que pertenece el objeto es conocida también por su identificador. Finalmente, se puede concluir que el objeto mostrado es un objeto del nivel base, dado que tiene asociado un meta-espacio.

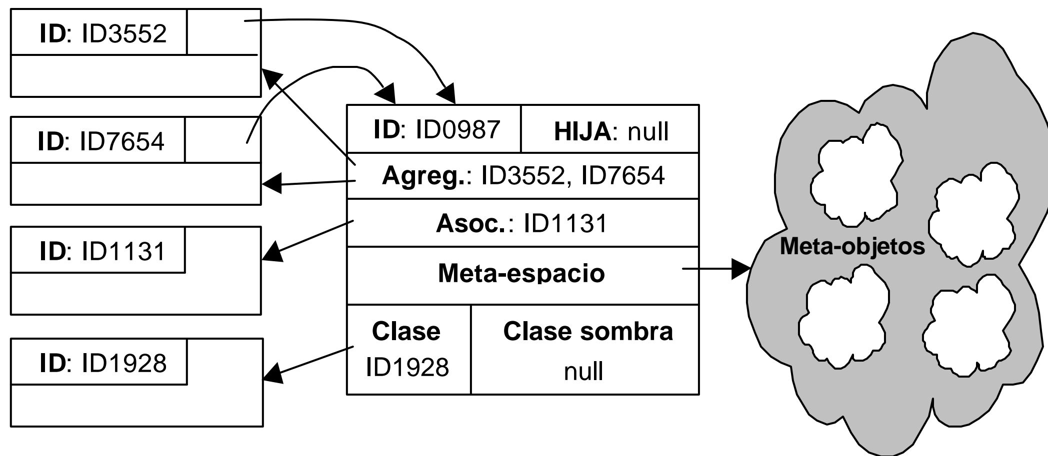


Figura 10.7. Representación interna de un objeto.

10.5.1 La clase sombra

La clase sombra juega un **papel determinante** para la distribución de los objetos. Su papel consiste en **proporcionar a los objetos la ilusión** de que la información que van a necesitar de su clase y superclases para realizar su computación está disponible de manera local para su utilización por la máquina abstracta.

La clase sombra es, en realidad, una **clase creada en tiempo de ejecución** que incluye una copia de la definición de los atributos y métodos de un objeto que no están disponibles en la máquina abstracta. Toda aquella información que sí está disponible localmente, no se copiará en la clase sombra.

La clase sombra de un objeto se creará, por tanto, en el momento en que un objeto no disponga de manera local de toda la información que necesita para su computación. Se distinguen dos situaciones en las que hay que estudiar si es necesario o no crear la clase sombra de un objeto: cuando se crea el objeto y cuando un objeto llega a un nodo como resultado de una operación de migración. Por otra parte, la clase sombra será eliminada cuando se elimine el objeto al que está asociada.

En la figura siguiente se muestra un **ejemplo de la necesidad de la clase sombra**. Un objeto de la clase `hija` se encuentra ubicado en un nodo del sistema integral en el que no residen las definiciones de su clase ni de su superclase (`padre`). La clase sombra juega el papel de dichas clases en el nodo en el que reside el objeto, de tal forma que el objeto pueda realizar de manera local su computación. Aunque en la figura la clase sombra incluya la definición de los métodos de la clase y superclase del objeto, esto no tiene por qué ocurrir en la realidad. Únicamente deben estar presentes en la clase sombra las definiciones de los métodos que estén en ejecución en el objeto.

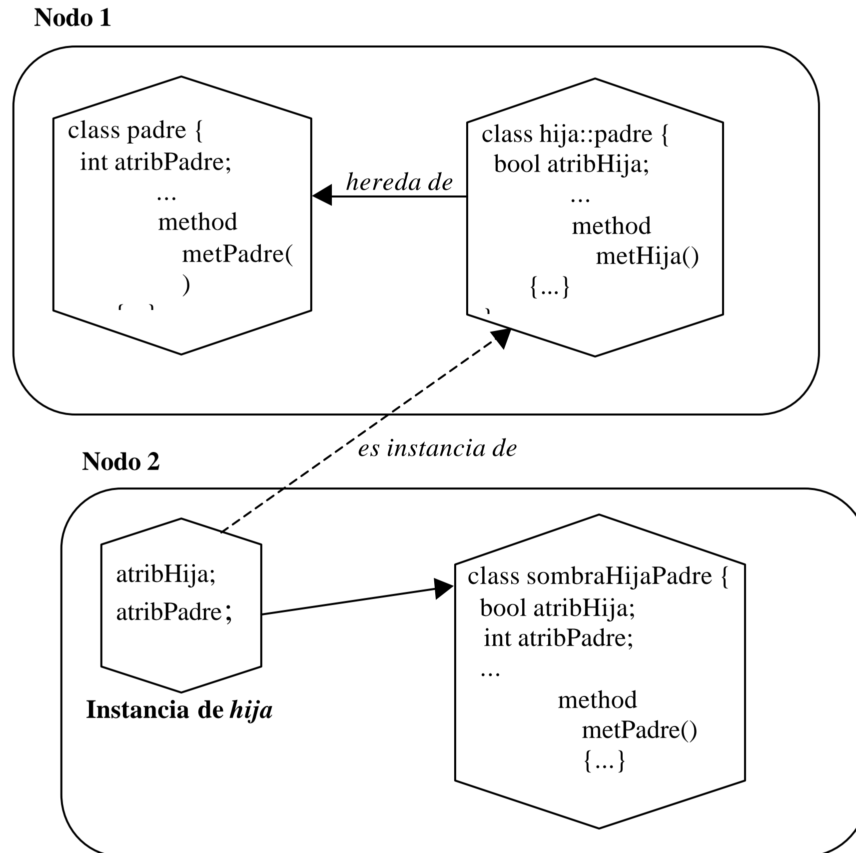


Figura 10.8. Necesidad de la clase sombra.

En los capítulos siguientes se describe en detalle cómo y cuándo es utilizada la clase sombra en la distribución de los objetos.

10.6 Aspectos semánticos de la distribución

La integración de la distribución en el sistema integral plantea un conjunto de **cuestiones semánticas** que tiene que ser tenidas en cuenta para su diseño.

En los apartados siguientes se inicia la discusión de la forma que va a tomar el sistema de distribución de objetos (AGRA) dentro del sistema integral. Se trata de aspectos semánticos y en ningún caso se describen los mecanismos del sistema.

En los próximos capítulos se describen en detalle los mecanismos y políticas diseñados para AGRA y que siguen la semántica adoptada en las secciones siguientes.

10.6.1 ¿Debe ser transparente la distribución?

La programación distribuida se puede simplificar notablemente haciendo que la **distribución sea transparente** para el programador. La forma en que se consigue este objetivo en el sistema integral es **aprovechar la característica de reflectividad de la máquina abstracta para proporcionar al programador una única facilidad de invocación de métodos, independiente de la ubicación del objeto invocado**. El programador no tiene por qué preocuparse de si el objeto invocado se encuentra en el mismo o en diferente nodo que el objeto invocador, de manera que no tiene que realizar acciones distintas para una invocación local y una remota. Para un gran número de

aplicaciones, hacer la distribución transparente y que aún así se puedan beneficiar de su existencia, redundará en una programación más simple y una mejora de su rendimiento.

Por ejemplo, en sistemas de distribución de objetos como Corba [OMG99] o DCOM [Mic98], a pesar de suponer un avance muy importante para la construcción de aplicaciones distribuidas y que se oculta una parte importante de los entresijos de la comunicación entre objetos, el programador tiene que preocuparse de saber cuándo una operación involucra objetos remotos y cuándo no, de manera que programará de una manera u otra en función de ello.

Existe, por otro lado, un conjunto de aplicaciones que nacieron porque existe la distribución y necesitan cierto control sobre la distribución para sacarle todos los beneficios posibles [Hut87]. Por ejemplo, un programa encargado del equilibrado de carga necesita, obviamente, control explícito sobre la ubicación de las entidades. El sistema Emerald [Jul88] proporciona al programador primitivas de migración de objetos, del tipo “mover objeto O al nodo X”, que le dan un control muy grande sobre la ubicación de los objetos.

El sistema integral proporcionará el **control de la distribución a través de varios objetos del sistema operativo**. El control se realizará, por tanto, a través del mecanismo común de invocación de métodos. Aquellos objetos de usuario que obtengan los permisos adecuados para invocar a los objetos del sistema operativo mencionados, obtendrán finalmente cierto grado de control de la distribución.

En el momento en que se hace disponible el control de la distribución a nivel del sistema operativo, los lenguajes de programación que se implementen en el sistema integral podrán hacer uso o no de dicho control, es decir, podrán proporcionar primitivas de control de la distribución u optar por ignorar ellos mismos su existencia.

10.6.2 Cuándo mover un objeto

Todo objeto está localizado en todo momento en un nodo del sistema integral (los objetos que se encuentran en el almacenamiento de persistencia no entran dentro de esta discusión). La migración de un objeto puede tener dos orígenes distintos: se solicita desde un objeto de usuario o el sistema operativo decide que es necesaria su migración.

10.6.2.1 Migración bajo demanda

Si el **programador solicita que el objeto se mueva** a otro nodo, el sistema operativo será el encargado de atender la petición. El sistema operativo decidirá, basándose en la información de que disponga sobre el estado del sistema, si se mueve el objeto inmediatamente o si se retrasa un tiempo hasta que el sistema se encuentre en un estado más apropiado para realizar el movimiento. En el apartado 10.6.3.1 se muestran algunos ejemplos en los que puede ser interesante realizar alguna operación de migración de objetos.

10.6.2.2 Migración automática

El **sistema operativo**, en otros casos, puede **tomar la iniciativa para mover objetos** dentro del sistema integral. Ya se han comentado anteriormente cuáles son los objetivos básicos de la migración de objetos. El sistema operativo, con su iniciativa, tratará en todo momento de maximizar el rendimiento global del sistema integral moviendo objetos entre diferentes nodos. Para ello, necesitará estudiar el estado del sistema y decidir qué objetos son susceptibles de ser movidos para mejorar su

funcionamiento. En este caso no hay peticiones por parte del programador para mover objetos.

Para cualquiera de los dos casos descritos, es preciso **diseñar unas buenas políticas de migración**, de tal forma que los movimientos de objetos no vayan precisamente en contra de su principal objetivo, que es mejorar el rendimiento global del sistema integral.

10.6.3 Semántica del paso de parámetros

La elección de la semántica del paso de parámetros en las invocaciones a métodos no es trivial en el sistema integral. Se parte de un modelo de objetos activo, que dota a los mismos de capacidad de cómputo, aparte de almacenar su estado (datos) interno. Las referencias a posiciones de memoria en los sistemas tradicionales se convierten en referencias a objetos en el sistema integral. Parece, en un principio, que el paso de parámetros por referencia es lo más natural en la invocación de métodos.

Para el caso del paso de parámetros por valor, sería necesario sacar una copia del objeto pasado como parámetro, copia que incluiría datos y estado del cómputo del objeto (estado de los hilos de ejecución internos al objeto). Se pierde, de alguna manera, el significado del paso por valor.

El **paso de parámetros se realizará siempre por referencia**. En el caso de invocaciones remotas, el paso por referencia puede ser ineficiente, dado que el acceso a los propios parámetros desde el objeto invocado se convertirá a su vez en un acceso remoto (ver figura siguiente).

Para determinado tipo de objetos, denominados **objetos inmutables** [LSA+77], sería posible, en cambio, el paso por valor. Un objeto inmutable es aquel que no puede cambiar a lo largo del tiempo. Para poder decidir, en un momento dado, si un parámetro se pasa por valor o por referencia, sería necesario saber si el objeto es o no inmutable. El modelo de objetos de la máquina no hace esa distinción, de manera que implícitamente supone que todos los objetos pueden cambiar su valor.

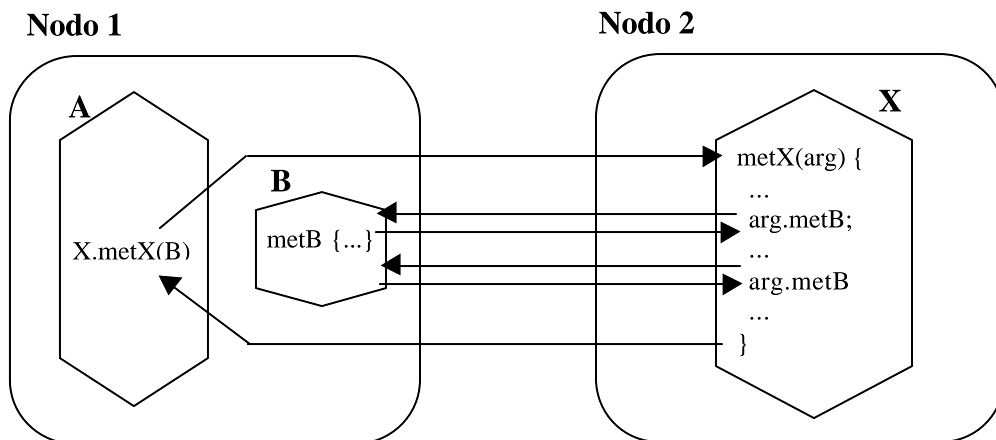


Figura 10.9. Problema del paso de parámetros por referencia.

10.6.3.1 Optimizaciones del paso de parámetros

Con el fin de mejorar las prestaciones del paso de parámetros por referencia, los lenguajes de programación podrán extender la semántica ofrecida por el sistema integral, de las siguientes maneras:

- Permitiendo al programador indicar qué parámetros se quieren mover al nodo en el que reside el objeto invocado. El lenguaje proporcionará al programador una palabra reservada que podrá acompañar a los parámetros de las invocaciones que se deseen migrar al nodo en el que reside el objeto invocado. En la figura siguiente, el objeto A del nodo 1 invoca al objeto X del nodo 2, pasándole como parámetro el objeto B, que inicialmente residía en el nodo 1, pero, como efecto de la invocación, migrará al nodo 2.

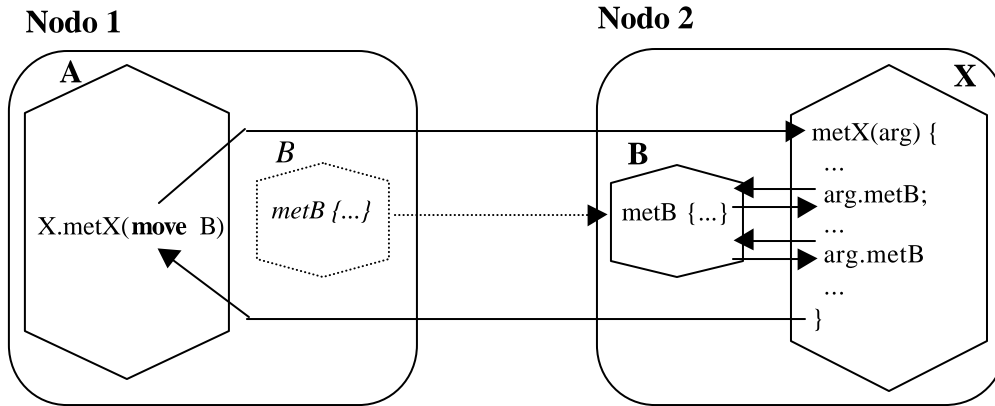


Figura 10.10. Migración de parámetros en una invocación remota.

- Permitiendo al programador indicar que el objeto apuntado por la referencia devuelta como resultado de la invocación se quiere mover al nodo en el que reside el objeto que la realizó. En la figura siguiente, el objeto A del nodo 1 invoca al objeto X del nodo 2, indicándole que el objeto resultado debe moverse al nodo 1.

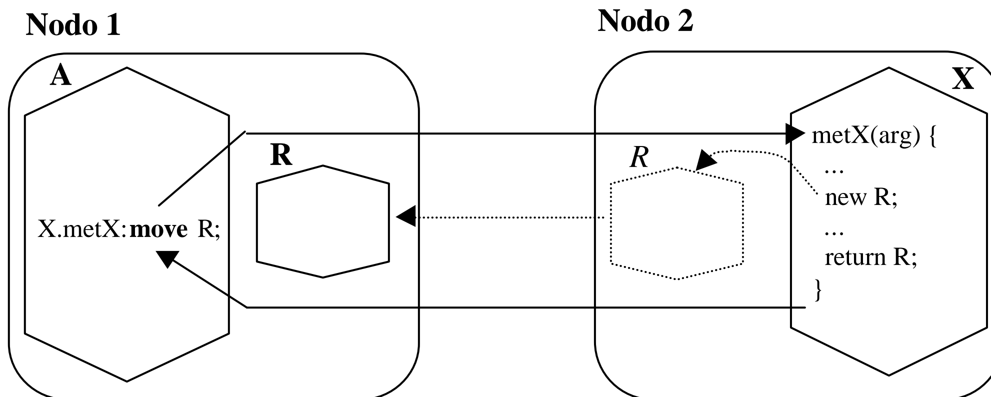


Figura 10.11. Migración del resultado en una invocación remota.

- Permitiendo al programador realizar cualquier otro tipo de indicación sobre el nodo de destino de los parámetros y/o resultado de una invocación, por ejemplo, para emular el paso de parámetros por valor. En la figura siguiente, el programador desea pasar el objeto B por valor, por lo que se saca un duplicado del objeto (B') que podría migrar al nodo 2 como ocurría en la primera situación de las descritas.

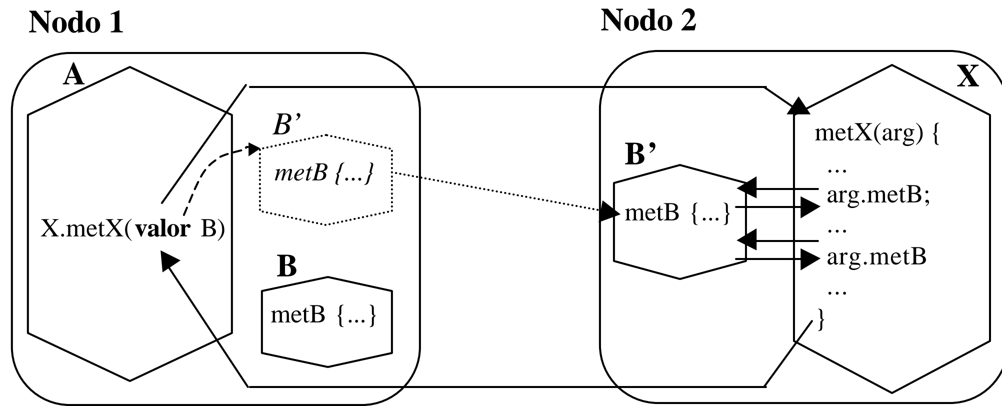


Figura 10.12. Paso de parámetros por valor.

En cualquier caso, la **semántica del paso de parámetros proporcionada por el sistema operativo es única** (paso por referencia). Se deja cualquier otro tipo de consideración, como las mostradas o como la declaración de un objeto como inmutable, para lenguajes u otras herramientas que deseen sacar un partido especial de la forma en que se manipulan los parámetros, de manera similar a como hace Emerald [Jul88].

10.6.4 Cuestiones relativas a la concurrencia

Con el fin de poder sacar el máximo partido a la distribución es necesario proporcionar al programador herramientas para el control de la ejecución y la sincronización. El control de la ejecución se puede estudiar a dos niveles: control de la ejecución de objetos diferentes y control de la ejecución de múltiples métodos dentro de un mismo objeto.

10.6.4.1 Control de la ejecución intra-objeto

Por defecto, la máquina abstracta proporciona a todos los objetos un mecanismo de control de la concurrencia basado en una ejecución de los hilos totalmente secuencial, es decir, cada hilo de ejecución deberá finalizar antes de permitirse la ejecución de otro hilo.

Es posible, sin embargo, sustituir el funcionamiento por defecto del mecanismo de control de la concurrencia, de tal forma que la política de ejecución de métodos (control de la concurrencia) viene determinada por meta-objeto del nivel del sistema operativo: el **meta-objeto sincronizador**.

El meta-objeto sincronizador define la política de aceptación de mensajes, estudiando las distintas restricciones de ejecución de un método antes de determinar si es o no posible. El estudio tendrá en cuenta el estado del entorno de ejecución del objeto:

- qué métodos están ya en ejecución
- qué métodos están a la espera de ejecutarse
- qué hilos ejecutando determinados métodos están esperando en una condición de sincronización.

La ejecución de métodos en el meta-objeto sincronizador se produce siempre como resultado de la ocurrencia de uno de los eventos siguientes: invocación de un método,

suspensión de un método, reanudación de un método o finalización de un método del objeto base asociado.

10.6.4.2 Control de la ejecución entre objetos distintos

El control de la concurrencia entre objetos distintos se consigue con dos mecanismos básicos: la invocación de métodos y el objeto primitivo cerradura.

La **invocación síncrona** de métodos (*call*) determina que el hilo de ejecución en el que se produce la invocación quede bloqueado hasta que el método invocado termine (devolviendo un resultado o una excepción).

En la **invocación asíncrona** (*wait-by-necessity*) el hilo de ejecución en el que se produce la invocación puede continuar su ejecución, produciéndose la sincronización cuando se desee utilizar el objeto resultado de la invocación al método. Trabajos como [CV98] muestran que la invocación asíncrona, a pesar de su mayor complejidad, supone una mejora del rendimiento, fundamentalmente en ambientes paralelos y distribuidos.

Finalmente, para aquellas aplicaciones que necesiten un control más afinado de la ejecución, la máquina abstracta proporciona la clase primitiva **cerradura** (*lock*), con la semántica tradicional [SG98].

10.7 Resumen

La introducción de la distribución en el sistema integral hace necesario ampliar el conjunto de características básicas definidas por el modelo básico de objetos proporcionado por la máquina abstracta. El nuevo conjunto de características únicamente tiene sentido en el caso de que exista la distribución de objetos y, por tanto, no puede venir predeterminado, aunque sí influenciado, por la máquina abstracta.

Los objetos del sistema integral distribuido seguirán un modelo de objetos activo. Se permitirán objetos de cualquier granularidad a efectos de su migración e invocación remota. Además, todos los objetos son potencialmente móviles e incluso persistentes. Sólo existirá una copia de cada objeto en todo el sistema integral. Finalmente, aquellos objetos que quieran proporcionar sus servicios al resto, habrán de publicarse en algún tipo de servicio de nombrado.

Otra decisión de diseño a tomar es la forma que van a tomar los objetos en tiempo de ejecución, es decir, de qué manera se relacionan entre ellos y se disponen en el área de instancias. En este aspecto, aparece el concepto de clase sombra como clave para posibilitar la distribución de objetos. La clase sombra reemplazará parcialmente a la jerarquía de clases para aquellos objetos que necesiten información sobre la clase a la que pertenecen y que no esté disponible localmente.

Finalmente, es necesario definir determinadas cuestiones semánticas relacionadas directamente con la distribución de objetos. La transparencia de la distribución se identifica como irrenunciable, aunque sea posible proporcionar cierto grado de visibilidad a los objetos que así lo necesiten. La forma en que se pasan los parámetros en las invocaciones a métodos es muy importante, ya que va a determinar la forma en que debe ser construida la facilidad de invocación remota. La existencia de múltiples máquinas abstractas con capacidad de cómputo complica la sincronización de tareas. Será necesario proporcionar al usuario herramientas para ello.

CAPÍTULO 11 GESTIÓN BÁSICA DE OBJETOS

11.1 Introducción

El paso de una arquitectura centralizada a una distribuida introduce una serie de problemas a resolver que se consideraban elementales en un entorno centralizado. La aparición de estos problemas viene dada, en general, por la **probable no disponibilidad de información completa** en ningún nodo del sistema integral para realizar incluso las operaciones más elementales. Otro tipo de problemas básicos a resolver surgen por la propia naturaleza distribuida del sistema integral y que en la versión centralizada no tenían sentido.

Destacan como operaciones elementales a resolver en un entorno distribuido de objetos, las siguientes:

- Nombrado y localización de objetos.
- Creación y eliminación de objetos.
- Comprobación de tipos en tiempo de ejecución.
- Interacción con la red de comunicaciones.

El nombrado de objetos y su localización en el sistema integral se presentan como fundamentales. Por un lado, se tiene que asegurar que los objetos se van a crear con un identificador único independientemente del nodo en el que se realice la operación de creación. Por otro lado, el resto de las operaciones enumeradas y otras que se verán más adelante necesitan de un mecanismo de localización de objetos que permita conocer la ubicación de un objeto involucrado en una operación y a partir de una información mínima, como puede ser su identidad.

Los problemas que se presentan para la creación y eliminación de objetos y la comprobación de tipos vienen dados por la dispersión de los objetos y la descripción de sus clases y superclases dentro del sistema integral, y se busca una solución que no obligue a repetir toda la información disponible en cada uno de los nodos. Sólo se puede hacer una excepción a esto último y es que toda la información sobre las clases básicas (primitivas) del sistema estará disponible en todos los nodos.

Finalmente, la existencia de una red de comunicaciones que conecta los diferentes nodos del sistema integral y la necesidad de su uso para permitir la interacción entre ellos obliga a dar una solución a la transmisión de información sin romper el modelo orientado a objetos.

En los apartados siguientes se realiza un **análisis en profundidad** de los problemas básicos que surgen al dotar de la característica de distribución al sistema operativo del sistema integral.

11.2 Nombrado de objetos

11.2.1 Identificadores y referencias

Tal y como viene determinado por la máquina abstracta subyacente, los objetos del sistema tienen asociado un **identificador único** que se les asigna en el momento de su creación y que no será reutilizado una vez que el objeto sea eliminado. Las diferentes máquinas abstractas que forman el sistema distribuido son capaces de crear identificadores diferentes para todos los objetos, de tal forma que se garantiza que no se van a repetir identificadores y que no se va a agotar el "espacio de identificadores" para la demanda futura de nuevos objetos. La forma de asegurar ambas cosas es utilizar un **algoritmo de generación de identificadores** que incluya la hora, fecha e identidad del nodo en el que se creó el identificador, de una manera similar a como lo hace DCE para crear los UUID (**ver capítulo de PANORÁMICA**).

El identificador único de los objetos es un nombre interno, de bajo nivel, que será utilizado, en general, por las máquinas abstractas para referenciar a los objetos, independientemente de su localización física. Dicho identificador será utilizado para identificar a un objeto, sea este local, remoto o se encuentre en el almacenamiento de persistencia, evitándose de esta manera la transformación de identificadores/punteros (*pointer swizzling*) [Álv98].

Para conocer el identificador de un objeto simplemente habrá que invocar el método *getID()* del mismo.

```
Class Object
Methods
   getID(): String;
   ...
Endclass
```

No hay que confundir el concepto de identificador con el de **referencia**. Esta última es utilizada para las operaciones de acceso a los objetos (invocación de métodos) y proporciona la única manera de conseguirlo. Las referencias contienen el identificador del objeto al que apuntan (al que hacen referencia) y determinada información de protección [Día2000].

El identificador único será asignado en el momento de creación del objeto, que mantendrá durante toda su existencia, independientemente de su ubicación, es decir, a pesar de que sea movido o que haya sido llevado al almacenamiento de persistencia. De la misma forma, las referencias a objetos tampoco cambian por el hecho de que cambie la ubicación del objeto que referencian. Las referencias no contendrán, por tanto, información alguna de localización, ni siquiera en la forma de pistas de localización que si existen en otros sistemas, como por ejemplo los sistemas operativos SOS [SGH+89] y Guide [BBD+91].

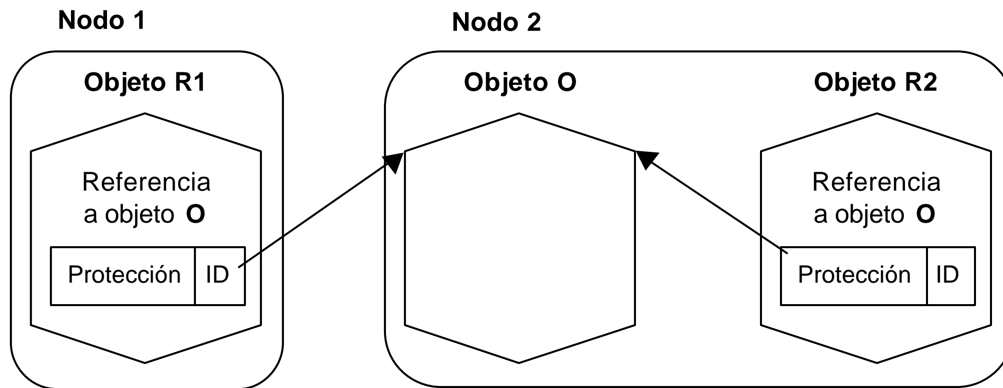


Figura 11.1. Referencias, identificadores e información de protección.

11.2.2 Nombres simbólicos

Aparte del mecanismo de nombrado de bajo nivel (asignación de un identificador) soportado por la máquina abstracta, es posible la implementación de **mecanismos de nombrado de alto nivel** que trabajen con nombres simbólicos y que estén soportados por el sistema operativo. Las tareas de traducción del sistema de nombrado de alto nivel al de bajo nivel serán realizadas por un conjunto de objetos del sistema operativo que compondrán lo que se denomina **servicio de denominación o servicio de nombrado**.

La utilización de nombres simbólicos para las entidades que maneja un sistema operativo es una característica presente en todos ellos. En los sistemas operativos tradicionales es difícil encontrar nombres simbólicos para entidades que no sean los ficheros. En los sistemas operativos orientados a objetos, los objetos, independientemente de su tipo, tamaño, etc., pueden tener asociado un nombre simbólico.

Lo que realmente obtendrán los objetos como resultado de una petición al servicio de denominación será una **referencia** al objeto buscado.

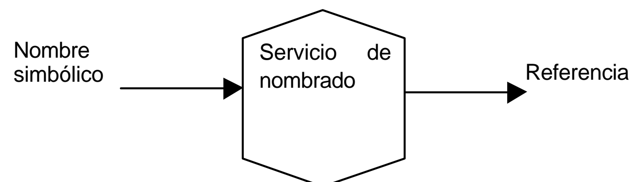


Figura 11.2. Utilización del servicio de nombrado

La obtención de una referencia a partir del nombre simbólico del objeto viene a jugar el papel de la llamada al sistema *open()* para los ficheros en los sistemas operativos UNIX. Se trata del primer paso a dar para estar en disposición de utilizar el recurso, objeto en el caso actual, fichero en el caso referido. Una vez superado este primer paso, el propietario de la referencia (el descriptor de fichero, en UNIX) está en condiciones de utilizar el objeto (fichero) a través del conjunto de métodos (llamadas al sistema) definido por su clase (definidas por el sistema operativo). Internamente, el sistema operativo diferenciará unos objetos (ficheros) de otros no por su referencia (descriptor), que podrá tener varias, ni por su nombre simbólico, que también podrá haber varios, sino por su identificador único de objeto (su *inodo*, en el caso de UNIX).

Obviamente, un servicio de denominación debe permitir registrar nuevos nombres simbólicos para objetos que así lo deseen.

```
Class ServiciodeDenominación
Isa object
Methods
   ObtenerReferencia(string):object;
   RegistrarReferencia(string, object);
   EliminarReferencia(string);
Endclass
```

11.2.3 Políticas de nombrado

El espacio de nombres (identificadores) de bajo nivel de los objetos del sistema integral no se puede cambiar, dado que las máquinas abstractas deben asegurar la propiedad de unicidad de los mismos en el espacio y en el tiempo. Es decir, el mecanismo de generación de identificadores para los objetos está integrado en la máquina abstracta y no puede ser reemplazado por ningún mecanismo del sistema operativo.

Las políticas de nombrado tienen sentido únicamente para los nombres simbólicos de los objetos. Al nivel del diseño del sistema de distribución básico del sistema integral, basta con saber que el servicio de denominación se encarga de convertir nombres simbólicos en referencias de objetos. La estructura que toma el servicio de denominación y, en particular, los nombres simbólicos, es indiferente para el sistema de distribución, dado que internamente trabajará con identificadores. De todas formas, y con el fin de acercarse a las posibles alternativas de implementación de un servicio de denominación, el capítulo 13 trata el tema en más profundidad.

11.3 Localización de objetos

La localización es esencial para la distribución de objetos. La localización de un objeto consiste en determinar su **ubicación** actual a partir de una información tan básica como puede ser su identificador (su nombre interno), información que se obtiene de cualquier referencia al objeto. En el caso de que el nombre suministrado para localizar el objeto sea un nombre simbólico, el servicio de denominación será el encargado de hacer la traducción para obtener la referencia, que será lo que se utilice en última instancia para localizar el objeto.

La localización de objetos en el sistema integral distribuido es compleja por las siguientes razones:

- Las referencias de los objetos no contienen información de localización.
- Los objetos se pueden mover en cualquier momento de un nodo a otro, ya sea en almacenamiento principal (memoria principal) o en almacenamiento secundario (de persistencia).

Parece difícil, por tanto, que un procedimiento de localización pueda devolver un resultado con una semántica tan fuerte como: "la ubicación devuelta es la que se

corresponde con la ubicación del objeto". Se mostrará como conseguir dicho resultado para aquellas operaciones que así lo necesiten.

11.3.1 Semánticas de localización

En general, se puede hablar de dos semánticas para la operación de localización:

- **Semántica débil** (conocimiento antiguo). La ubicación devuelta es una en la que el objeto residió en algún momento en el pasado. La ubicación devuelta no deja de ser una **simple pista**, y puede estar obsoleta.
- **Semántica fuerte** (conocimiento reciente). La ubicación devuelta es una en la que el objeto ha residido en algún momento después de la iniciación de la petición de localización. Si además se asegura que el objeto no se ha movido después de haber preguntado su ubicación, se conocerá su **ubicación exacta**.

La localización de un objeto utilizando semántica débil se denomina **localización débil**; con semántica fuerte, se obtiene la **localización fuerte**.

Teóricamente, la localización débil podría serlo tanto que valdría con devolver la ubicación en la que fue creado el objeto. En la práctica, es más razonable devolver la última ubicación conocida. Computacionalmente es bastante barato, dado que sólo involucraría la inspección de estructuras de datos locales (no se realizarían consultas remotas). La ubicación obtenida es potencialmente obsoleta, aunque a menudo se puede considerar bastante precisa, dado que los objetos tienden a moverse con menos frecuencia que con la que son accedidos (invocados).

La localización fuerte implica que se verifique la ubicación actual del objeto en cuestión, verificación que sólo puede ser proporcionada por el nodo en el que realmente reside el objeto. Esto puede requerir una búsqueda costosa en el sistema integral distribuido. Incluso si el objeto no se ha movido desde la última vez que se supo de él, será necesario preguntar a su nodo.

En la tabla siguiente se comparan los dos tipos de localización. Las ventajas principales de la localización débil son su rapidez y bajo coste; su principal desventaja es que puede fallar al intentar proporcionar una respuesta útil. La principal ventaja de la localización fuerte es que la ubicación proporcionada está al día; sus principales desventajas son su lentitud y coste.

Característica	Localización débil	Localización fuerte
Semántica	Pista	Actualizada
Implementación	Usa información local	Verificar la pista o búsqueda
Eficiencia	Muy alta	Muy baja
Velocidad	Muy alta	Muy baja
Información al día	Baja	Alta

Tabla 11.1. Características de la localización débil y fuerte.

La localización fuerte es imprescindible en un sistema integral distribuido y tendrá que tener en cuenta la posibilidad de que alguno de los nodos del mismo no estén disponibles. Se puede añadir, por tanto, a la definición de la semántica de la localización fuerte lo siguiente: "cuando una petición de localización falla, el objeto no es accesible

en ningún nodo de los accesibles en el momento en que se completó la operación de localización".

11.3.2 Localización de objetos en el sistema integral

La localización de objetos es imprescindible en diversas operaciones que tienen que ver con la distribución del sistema integral y que son realizadas a nivel del sistema operativo:

- **Invocación de un objeto**, dado que es necesario conocer el nodo en el que reside el objeto invocado para enviarle los parámetros de la invocación.
- **Devolución del resultado/excepción** de una invocación. Cuando finaliza la ejecución de un método, es necesario devolver el resultado del mismo. Es preciso localizar el objeto al que devolver el resultado, dado que puede haberse movido.
- **Migración de un objeto** para, por ejemplo, localizar un conjunto de objetos relacionados con el fin de que se muevan al mismo nodo.
- **Encontrar el tipo de un objeto**, con el fin de realizar operaciones de comprobación de tipos.

Para aquellas aplicaciones que se construyen con el conocimiento de la existencia de distribución en el sistema integral la operación de localización es muy importante, dado que basan ciertas decisiones de su computación en la ubicación de determinados objetos en instantes concretos. Por ejemplo, aquellas aplicaciones que necesiten maximizar el rendimiento en la utilización de un determinado recurso, pueden solicitar su migración al nodo en el que reside este.

La información devuelta por una petición de localización de un objeto será una referencia a un objeto de la **clase nodo**, que encapsula la información necesaria para identificar un nodo del sistema integral.

En general, la localización fuerte es la más recomendable, dado que devuelve siempre información actualizada. Como inconvenientes están su coste y lentitud. Usar siempre localización fuerte lleva consigo una gran sobrecarga del sistema, dada la frecuencia con la que es solicitada una operación de localización.

La arquitectura de la máquina abstracta permite **obtener los beneficios de la localización fuerte con el coste de la localización débil** para las instancias locales, dado que una operación de búsqueda de un objeto en el nodo local es una búsqueda muy rápida, puesto que sólo involucra la consulta de estructuras de datos locales y no se producen retardos de comunicaciones. El rendimiento de las operaciones de localización podrá entonces maximizarse siempre que se asegure que la mayor parte de los objetos buscados sean locales. La **utilización de políticas de migración de objetos adecuadas** permitirá conseguirlo.

En definitiva, se utilizará localización débil como primer paso para localizar un objeto. Si con la localización débil se dispone de la información suficiente para llevar a cabo la operación, se habrá conseguido el ahorro de realizar una localización fuerte. Si la localización débil fracasa y es imprescindible obtener información de localización fiable, se realizará un segundo paso de localización con semántica fuerte.

11.3.2.1 Objetos del sistema operativo para la localización

Para facilitar la localización de un objeto y la implementación de múltiples políticas de localización, el sistema operativo proporciona un objeto denominado **localizador**, que será el responsable de retornar al objeto que así lo solicita la información de localización de un objeto, dado su identificador o referencia.

El objeto localizador es la parte visible del sistema de localización, pero no necesariamente la única. En general, se supone que el sistema de localización dispone en cada nodo del sistema integral de un objeto localizador, que actúa como interfaz para los objetos de usuario, y un conjunto de **objetos de localización secundarios** que implementan las posibles políticas de localización. Con el fin de maximizar el rendimiento de las operaciones de localización, los objetos de localización secundarios registrarán la información necesaria (o posible) sobre la ubicación de objetos no locales que son invocados desde objetos locales.

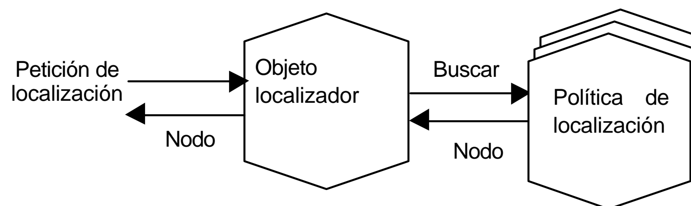


Figura 11.3. Relación entre el objeto localizador y la política de localización.

La existencia de objetos localizador en todos los nodos del sistema integral permite que el máximo número de solicitudes de localización puedan ser resueltas de manera local, siempre y cuando implementen buenas políticas de localización para registrar la actividad de los objetos locales con objetos remotos. Adicionalmente, y con el mismo objetivo, los distintos objetos localizador podrán dialogar entre ellos para intercambiar información de localización.

Objetos locales

Dado que el coste mínimo de localización de un objeto se consigue cuando el objeto a localizar reside en la máquina abstracta en que se solicita la operación, las diferentes políticas de localización comenzarán la búsqueda de un objeto en la máquina abstracta local. Esta operación es realizada internamente por la máquina abstracta, buscando en su área de instancias. Si el objeto buscado reside en la máquina local, la operación de localización finaliza inmediatamente, con un **coste muy bajo**, y con el resultado de una localización fuerte (se tiene total certeza sobre la ubicación del objeto) aunque no hubiera sido solicitada de esa manera.

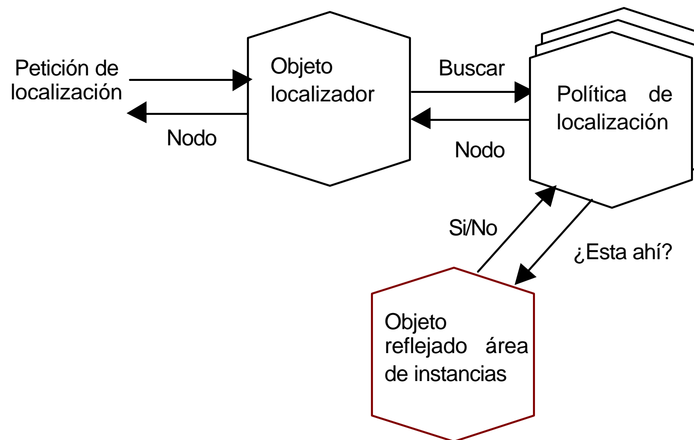


Figura 11.4. Búsqueda del objeto en el área de instancias local.

Objetos remotos

En el caso de que el objeto invocado no se encuentre en la máquina local, hay tres posibilidades más: que esté en el almacenamiento de persistencia, que esté ubicado en algún otro nodo del sistema integral o que ya no exista. Diferentes políticas de localización utilizarán diferentes estrategias para obtener la localización de un objeto.

11.3.2.2 Política de localización por defecto

El sistema de distribución de objetos incorpora una política de localización por defecto para los objetos localizador, que se va a utilizar de aquí en adelante. La política de localización por defecto es una **política de localización débil**.

Con el fin de posibilitar la operación de localización, el objeto localizador utilizará una **cache de direcciones de objetos no locales**, que recogerá información de la identidad de objetos para los que existen referencias en el nodo local pero que no se encuentran en el área de instancias local. Un esquema de cache similar se puede encontrar en el sistema operativo Amoeba [TRS+90, MRT+90].

La cache **almacenará información para identidades de objeto** y no para referencias, dado que un mismo objeto puede ser referenciado por múltiples referencias distintas de un nodo.

La información a almacenar para cada identidad de objeto dará alguna **pista** sobre la ubicación del objeto: un nodo remoto, el almacenamiento de persistencia o si se sabe a ciencia cierta que el objeto ha sido eliminado. Obviamente, dado que los datos que se guardan en la cache solamente se pueden utilizar como pistas para localizar objetos, el hecho de que un objeto no esté en la cache no presupone que el objeto ya no exista en el sistema integral, es decir, que haya sido eliminado. Debido a la limitación en tamaño de la cache, ésta tendrá que implementar una **política de remplazo** de direcciones de objetos, que, por defecto, es una política LRU.

En la figura siguiente se muestra una posible implementación de la cache de direcciones de objetos remotos. La cache se organiza como una tabla de dos entradas: la primera, almacena el identificador de un objeto; la segunda, registra la información de ubicación disponible para el objeto correspondiente.

ID	ubicación
AFD87	Nodo1
XZP09	Persist.
JJK321	Borrado
...	...

Figura 11.5. Cache de direcciones de objetos no locales.

La **política de localización por defecto** utiliza el área de instancias de la máquina local y la cache de direcciones de objetos no locales para resolver una petición de localización. Cuando se solicita obtener la localización de un objeto, se consulta en primer lugar el área de instancias de la propia máquina abstracta. Si se encuentra en ella, la operación de localización finaliza. Si no se encuentra, se pasa a consultar la cache de direcciones. La consulta de la cache de direcciones puede tener éxito o no. En caso de tener éxito, el objeto buscado está registrado en la cache, por lo que se devuelve la información de localización asociada al mismo. Si la consulta fracasa, se devuelve una excepción LOCATION_NOT_FOUND.

Como se puede comprobar, la política de localización por defecto utiliza únicamente estructuras de datos locales a la propia gestión de la operación. Su coste es, por tanto, bastante bajo y se corresponde, como se ha comentado, con una localización débil.

11.3.3 Opciones de localización

Si se desea realizar una operación de **localización con semántica fuerte**, no basta con mirar en la cache de direcciones de objetos no locales. Incluso en algunos casos, no basta con conocer la ubicación exacta de un objeto, sino actuar en función de ella.

Con el fin de poder ir más allá de la simple obtención de información de localización de un objeto, existen las **opciones de localización**. El objeto localizador establece un comportamiento por defecto ante la ausencia de toda opción (la semántica débil comentada en el apartado anterior), que puede modificarse con la adición de una o varias de ellas. Las opciones disponibles son las siguientes:

- **Fijar**. Con esta opción, la operación de localización del objeto lo etiqueta como inmóvil en el nodo en el que se encuentre, siempre y cuando el objeto estuviera activo, dado que si se encontrase en el almacenamiento de persistencia se considera que no se encuentra en nodo alguno.
- **Liberar**. Libera a un objeto de la atadura creada con una operación fijar.
- **Asegurar**. Esta opción se utiliza para permitir la localización fuerte, para aquellos casos en los que la ubicación del objeto es no local.

La opción fijar se utiliza para hacer inmóvil un objeto activo en la ubicación en la que se encuentre. Se presenta como una utilidad para controlar la ubicación de los objetos, de tal forma que cualquier objeto puede pasar de ser potencialmente movible a inmóvil y viceversa. Para que un objeto quede fijo en un nodo, basta con que se modifique el atributo `móvil`, de tipo lógico, de su meta-espacio, almacenado

concretamente en su meta-objeto datos. Si este atributo tiene un valor `falso`, el objeto no podrá elegirse para ser movido.

La opción liberar proporciona de nuevo a un objeto la capacidad de moverse por el sistema integral en el momento en que sea elegido. Su uso permite modificar el atributo `móvil` para que tome el valor `verdadero`.

Finalmente, la opción asegurar sirve para forzar una localización fuerte. La localización fuerte es necesaria en ciertos momentos en los que no basta con tener una idea aproximada de la ubicación de un objeto. El siguiente apartado trata en detalle la localización fuerte.

11.3.4 Localización fuerte

En algunos casos es necesario obtener información totalmente fiable de la ubicación de un objeto. Por ejemplo, en la operación de invocación de un objeto, es necesario conocer con total certeza en qué nodo está ubicado, para dirigirle el mensaje que contiene los datos de la invocación.

Como ya se ha comentado, la localización fuerte se debe utilizar como último recurso, dado que su coste puede ser muy elevado con respecto a una localización débil.

El primer paso, ya descrito, de la localización fuerte consiste en una búsqueda dentro del área de instancias de la máquina abstracta en la que se solicitó la operación. En el caso de que haya éxito, la operación de localización habrá concluido. En el caso de que se fracase, será necesario inspeccionar el resto de los nodos del sistema integral distribuido.

La **localización distribuida es realizada de manera cooperativa** por los objetos localizador ubicados en todos los nodos del sistema integral. El objeto localizador local pedirá a todos los objetos localizador remotos que realicen una búsqueda del objeto en su área de instancias local. Aquel objeto localizador remoto que lo encuentre contestará afirmativamente, de modo que el objeto localizador local conocerá su ubicación y podrá actualizarla en la cache de direcciones de objetos no locales. En la figura siguiente se muestra cómo se realiza todo el proceso, suponiendo que la operación de localización comienza en el nodo etiquetado con el número 1.

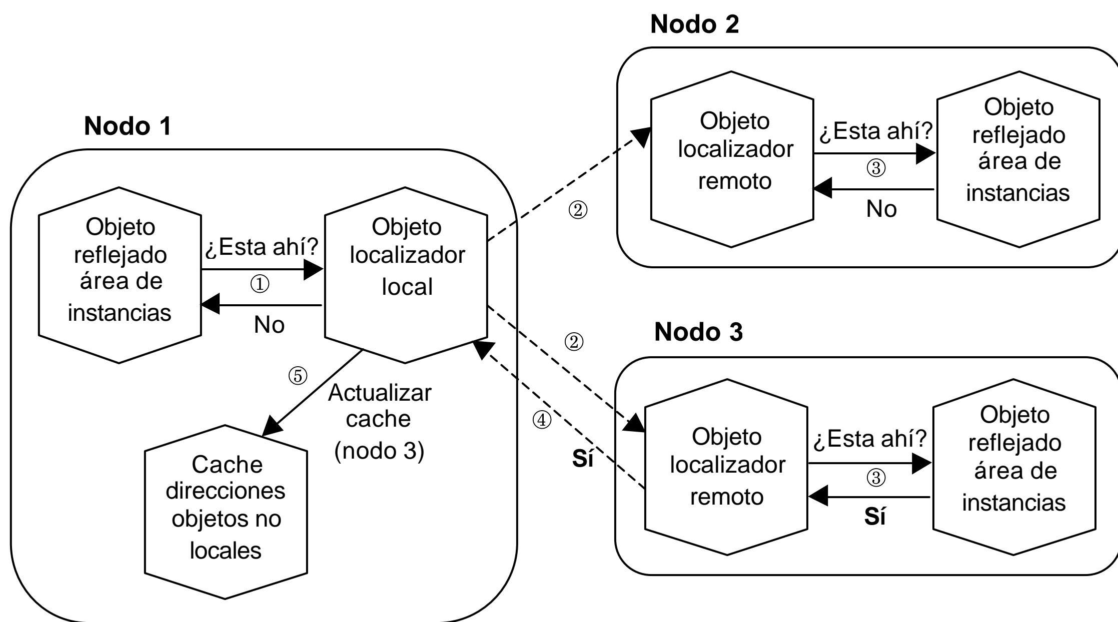


Figura 11.6. Esquema de localización fuerte en el sistema integral distribuido.

Si el objeto es encontrado pero se esté moviendo en ese momento a otro nodo del sistema integral, el localizador remoto informará de este hecho al localizador local, suministrándole además la identidad del nodo destino de la migración del objeto. La información que se actualizará en la cache de direcciones de objetos no locales será, por tanto, la correspondiente a este último nodo.

En el caso de que el objeto no se encuentre en ningún nodo, se devolverá una excepción INSTANCE_NOT_FOUND al objeto que solicitó la operación que provocó a su vez la realización de la operación de localización.

Por otro lado, si la operación de localización había sido solicitada con la etiqueta adicional para fijar el objeto, el objeto localizador remoto que hubiese encontrado el objeto lo fijará en el nodo.

11.3.5 Otras políticas

Las políticas descritas anteriormente se corresponden con las que, por defecto, están implantadas en el sistema de distribución del sistema integral. La implantación definitiva del sistema de distribución puede modificar dichas políticas para adaptarlas al entorno en el que se va a ejecutar.

Por ejemplo, una implantación determinada del sistema de distribución puede decidir sustituir el algoritmo LRU de sustitución de referencias en la cache de direcciones de objetos no locales por otra más adecuada.

Para el caso de la localización distribuida, los objetos localizador remotos podrían realizar una búsqueda del objeto en su cache, además de realizarla en su área de instancias. También para el caso de la localización distribuida, se puede pensar en que sólo conteste al objeto localizador local aquel localizador remoto que conozca con total certeza la ubicación del objeto buscado. El resto de las consultas no serían contestadas.

11.4 Creación y eliminación de objetos

En un entorno centralizado (una única máquina abstracta) las operaciones de creación y eliminación de un objeto son triviales dado que toda la información necesaria está disponible localmente: el área de instancias, en la que colocar el objeto a crear o de la que será eliminado el objeto, el área de clases, donde se pueden encontrar la descripciones de las clases y superclases del objeto, y el área de referencias, en la que se encuentra la referencia a utilizar para la operación.

En un entorno distribuido, la información mencionada está, generalmente, dispersa. Para el caso de la creación de un nuevo objeto, el caso extremo es aquel en el que las clases y superclases del mismo están ubicadas en diferentes nodos del sistema. En la figura siguiente se muestra un ejemplo de este caso. Una clase `nieta` hereda de una clase `hija` que, a su vez, hereda de una clase `padre`, y ninguna de ellas se encuentra en el mismo nodo.

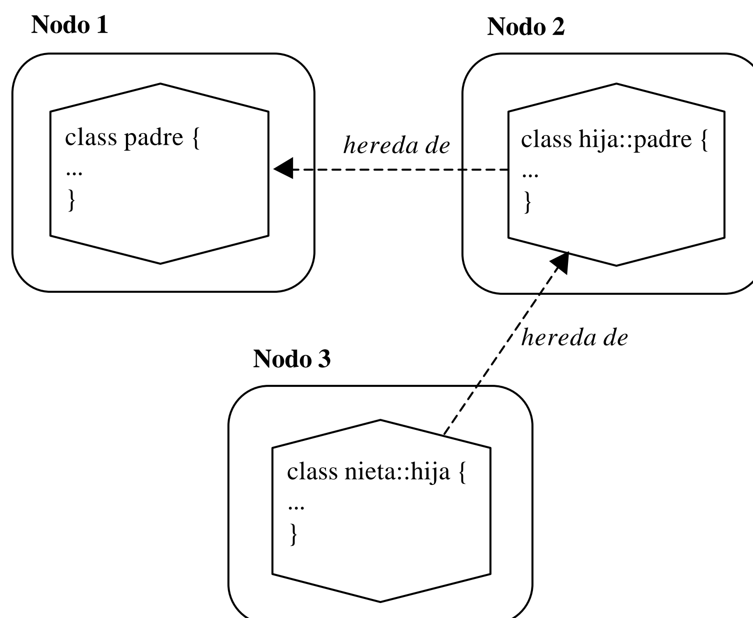


Figura 11.7. Problemática de la dispersión de clases para la creación de objetos.

El caso de la creación de objetos tiene una problemática adicional: dado que la información necesaria para la creación no estará, posiblemente, disponible de manera local, ¿ en qué nodo se crea el objeto? La respuesta a esta pregunta no es trivial, dado que sería necesario evaluar el coste de la creación local del objeto para una, muy posible, utilización (invocaciones) local en comparación con el coste de la creación y utilización remotas. Este tipo de decisiones determinan la necesidad de implantar **políticas de creación de objetos**.

Para la eliminación de un objeto, podría darse el caso de que la referencia utilizada para eliminarlo estuviera en un nodo, la instancia en otro nodo y alguno de sus atributos en un tercer nodo, tal y como se muestra en la figura siguiente, donde el objeto `principal` es el objeto a eliminar.

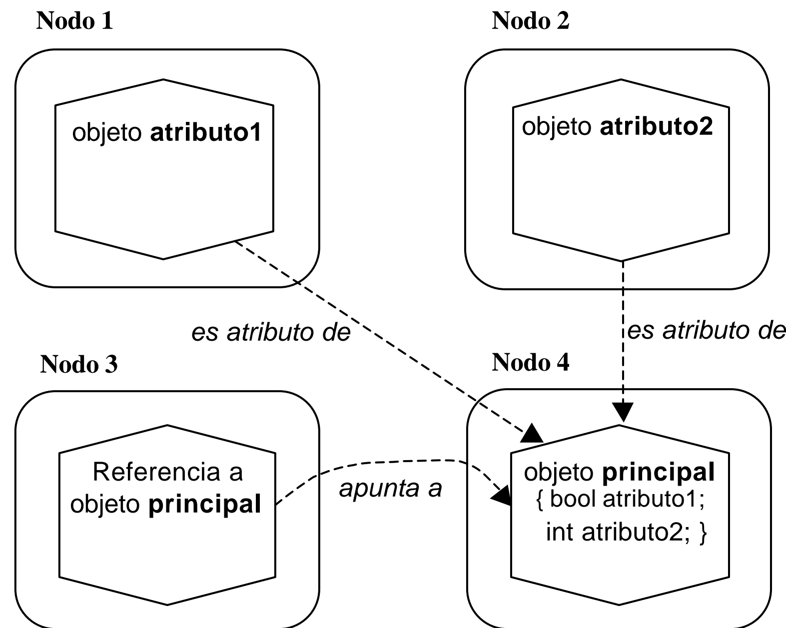


Figura 11.8. Situación extrema de eliminación de un objeto.

Para cualquiera de las situaciones descritas anteriormente es necesario tener en cuenta que las máquinas abstractas van a tener que solicitar frecuentemente la ayuda del sistema operativo para realizar las operaciones de creación y eliminación de objetos, con el fin de poder superar las fronteras que las separan.

Las operaciones de creación y eliminación de objetos implantadas por las máquinas abstractas son sobrescritas por el meta-objeto del sistema operativo denominado **gestor del ciclo de vida**. Este meta-objeto permitirá extender dichas operaciones al ámbito del sistema integral distribuido y podrá ser adaptado, como todos los meta-objetos, para cumplir con los requerimientos del diseñador del sistema operativo.

11.4.1 Creación de un objeto

La operación de creación de un objeto tiene como resultado final la **inserción de una nueva instancia (o varias) en el área de instancias** de alguna (o varias) de las máquinas abstractas del sistema integral y su asignación a una referencia que apuntará al objeto para su utilización. Dado que la operación puede requerir de información almacenada en varias máquinas abstractas, el sistema operativo la modifica de manera reflectiva, es decir, proporciona una implementación que sobrescribe la que realizan por defecto las máquinas abstractas.

La creación de la instancia y su colocación en el área de instancias consiste, básicamente, en asignar valores a los datos que se muestran en la figura siguiente.

Identificador de objeto
Referencia a objeto (instancia) del que es subobjeto (subinstancia)
Array de referencias a objetos agregados
Array de referencias a objetos asociados
Referencia al meta-espacio
Referencia a clase
Referencia a clase sombra

Figura 11.9. Estructura interna de una instancia.

El identificador del objeto se asigna de manera automática, y se corresponde con un valor único en el espacio y en el tiempo en el sistema integral. En el que caso de que el objeto sea un agregado (subobjeto) de otro objeto, será necesario almacenar dicha relación. Para aquellos objetos que tengan entre sus atributos algún agregado, la operación de creación deberá crear, recursivamente, los objetos agregados. Las referencias a los objetos asociados no toman valor inicial; el programador deberá darles valor explícitamente. La referencia al meta-espacio apuntará al mismo conjunto de meta-objetos al que apunta la referencia correspondiente en el objeto que demanda la operación de creación. De todas formas, es posible definir un **constructor especial** (del sistema operativo) que defina cómo se tiene que construir el meta-espacio de un objeto nuevo. También toma valor inicialmente la referencia a la clase a la que pertenece el objeto. Finalmente, y en determinados casos (se describirán a continuación), el objeto se creará con una clase sombra no vacía.

11.4.1.1 Creación local

Si el objeto a crear es de una clase primitiva, la creación será completada localmente por la propia máquina abstracta en la que se solicita la operación. Además, no será necesario crear la clase sombra, dado que las definiciones de todas las clases primitivas son conocidas en todas las máquinas abstractas del sistema integral.

Si el objeto a crear es de una clase definida por el usuario, pero toda la información sobre la clase y la de sus agregados está disponible localmente, la operación también será completada por la propia máquina abstracta. En este caso, tampoco será necesario crear la clase sombra.

11.4.1.2 Creación distribuida

En el caso de que no se disponga de toda la información necesaria, la máquina abstracta no podrá completar la operación por sí misma. La falta de información puede venir dada por la no disponibilidad, de manera local, de la definición de la clase, de alguna de sus superclases o por la ausencia de información equivalente de alguno de sus agregados.

Con el fin de simplificar la descripción de la creación de objetos distribuida, se va a suponer que la **política de creación por defecto** determina que un objeto se creará en el nodo en el que reside la clase a la que pertenece.

El conjunto de pasos a dar para realizar una operación de creación es el siguiente:

1. Localizar la clase a la que pertenece el objeto a crear. Se utilizará el objeto localizador que, con una política de localización fuerte, determinará el nodo en el

que reside la clase. A dicho nodo se envía un mensaje que contiene el identificador de la clase.

2. Dicho mensaje es procesado por un objeto trabajador, que realizará la parte local de la operación y solicitará, de manera remota, aquellas partes que no hayan podido ser resueltas localmente.
3. El objeto trabajador lanza, a su vez, un proceso de creación de los objetos agregados descritos en la clase, proceso que sigue los pasos aquí descritos para cada uno de ellos. Paralelamente, el objeto trabajador inicia un proceso de creación de la parte heredada de la clase localizando sus ancestros y enviando a los nodos en los que residen los mensajes necesarios para dicha creación.
4. Cuando la creación de un objeto se puede completar de manera local, se inserta en el área de instancias y se retorna su referencia como resultado, finalizándose esta rama de la creación recursiva.
5. Cuando ha finalizado completamente la creación de un objeto, se inserta en el área de instancias (a través de su objeto reflejado) y se retorna su referencia como resultado. En el caso de que el objeto fuera un atributo heredado, se devolverá también el identificador de su clase.

El objeto trabajador creado en el nodo en el que reside la clase del objeto a crear recibirá toda la información necesaria para insertar definitivamente el objeto en el área de instancias. Dado que la información sobre sus superclases se encuentra dispersa en el sistema integral, se creará su clase sombra, que incluirá una relación de todos sus atributos heredados (con sus tipos) y que pertenecen a clases que no están definidas en el nodo local.

Dada la complejidad de la operación de creación distribuida, se muestra a continuación su funcionamiento con un ejemplo. Se trata de crear un objeto de una clase `hija`, que hereda de una clase `padre`. La operación de creación es solicitada en el nodo 1, la definición de la clase `hija` está almacenada en el nodo 2 y la de la clase `padre`, en el nodo 3. Los atributos son de tipos habitualmente primitivos, con el fin de simplificar el ejemplo.

En primer lugar, se localiza el nodo en el que reside la clase hija y se le envía un mensaje para que se cree la instancia (1). El objeto trabajador estudia la definición de la clase hija (2) y pasa a crear los objetos correspondientes a los agregados de la clase (3a), a la vez que localiza y pasa a crear la parte heredada de la misma (3b). Para la parte heredada, ocurre algo similar. El objeto trabajador estudia la clase padre, crea los objetos agregados y pasa a crear la parte heredada (4b y 5b). Una vez que ha sido creada la parte heredada en el nodo 3, se devuelve al objeto trabajador del nodo 2 una lista con las referencias de los objetos creados para la parte heredada. Estas referencias y las creadas de manera local en el nodo 2 formarán la instancia a registrar en el área de instancias (6). Finalmente, se crea la clase sombra, que permite conocer la composición del objeto a partir de la información recogida de su clase y superclases. La clase sombra no incluye la definición de ninguno de los métodos del objeto, dado que aun no ha comenzado su actividad. La definición de los métodos se irá obteniendo, como se verá en el capítulo siguiente, de manera perezosa cuando se solicite la ejecución de los mismos.

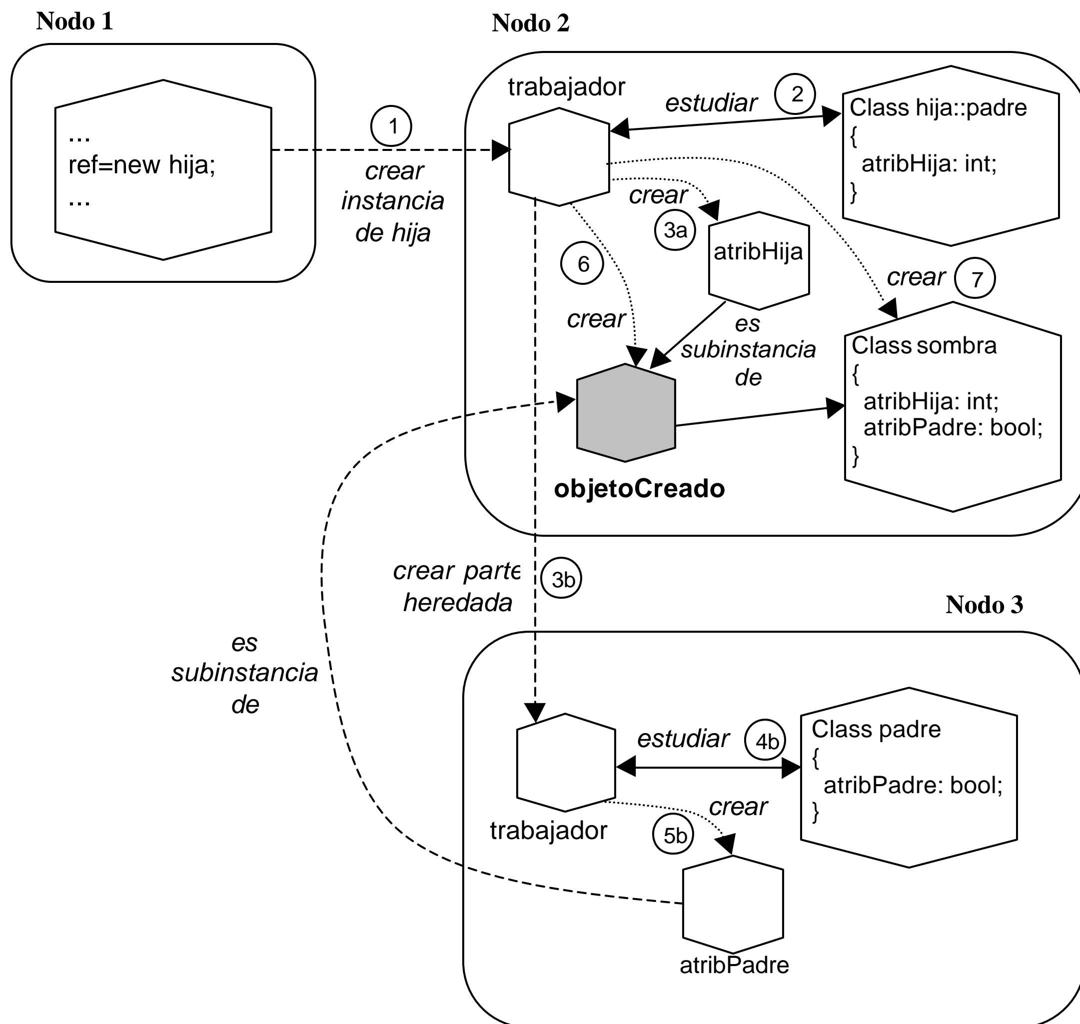


Figura 11.10. Creación distribuida de un objeto.

11.4.2 Eliminación de un objeto

Cuando un objeto deja de ser necesario, puede ser **eliminado de manera explícita**. De una manera similar a como ocurría con la operación de creación, la operación de eliminación liberará el espacio ocupado por el objeto y todos sus atributos agregados. Así mismo, y en el caso de que exista, se liberará el espacio ocupado por la clase sombra del objeto.

Dado que un objeto y sus agregados no están obligados a coexistir en el mismo nodo del sistema integral, la operación de eliminación puede expandirse por varios de ellos, eliminándose a su vez los agregados y las clases sombra de cada uno de los objetos que se vayan encontrando al seguir las referencias. De nuevo el sistema operativo colabora con la máquina abstracta para hacer posible la eliminación distribuida.

Al final del proceso, se habrá liberado el espacio ocupado por el objeto, sus agregados, los agregados de sus agregados, y así sucesivamente, y la referencia utilizada para la operación de liberación quedará libre (no apuntará a ningún objeto).

De nuevo es preciso la participación de objetos trabajador que se encargarán de realizar aquellas partes de la operación de eliminación del objeto que no se pueden

completar de manera local. A continuación se muestran los pasos a seguir para la eliminación distribuida de un objeto:

1. Localizar el objeto a eliminar a partir de la referencia de que se dispone. Si el objeto es local a la referencia, continuar en el paso 2. Si el objeto no es local, se envía un mensaje al nodo en el que reside para que se proceda a eliminar en él.
2. Obtener la lista de atributos agregados del objeto y aplicarles el algoritmo descrito por este conjunto de pasos.
3. Eliminar el objeto del área de instancias.

En la figura siguiente se muestra un ejemplo de eliminación de un objeto siguiendo los pasos anteriores. La operación de eliminación es solicitada en el nodo 1, estando ubicado el `objeto` a borrar en el nodo 2 y sus atributos (`atributo1` y `atributo2`) en los nodos 3 y 4. Desde el nodo 1 se envía el mensaje para la eliminación ① al nodo 2. El objeto trabajador obtiene las referencias a los atributos ② y envía mensajes para su eliminación (③ y ⑥). En el nodo 3, se procede a eliminar `atributo1` (④ y ⑤) y en el nodo 4, se hace lo mismo con `atributo2` (⑦ y ⑧). Finalmente, se elimina el objeto en el nodo 2 ⑨.

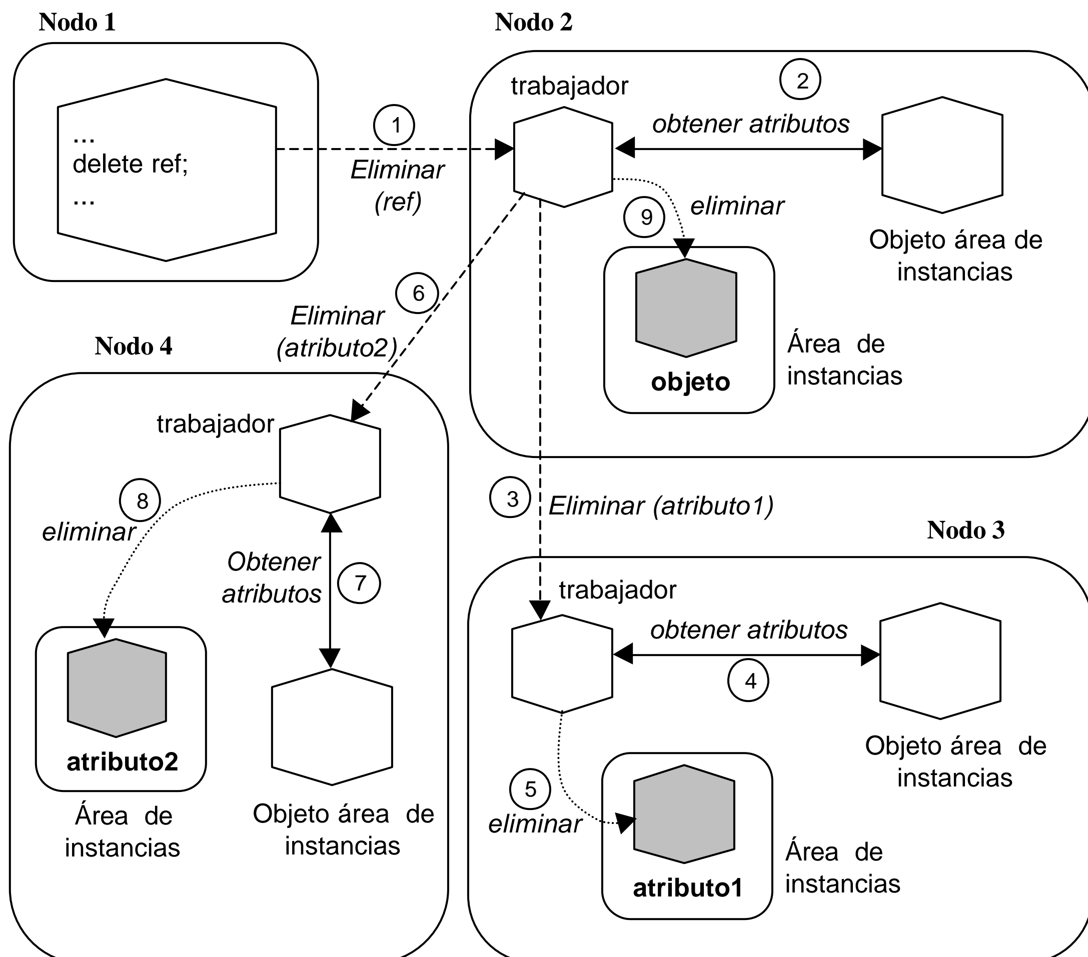


Figura 11.11. Eliminación de un objeto.

11.5 Comprobación de tipos

La comprobación de tipos es imprescindible en varias situaciones del funcionamiento del sistema integral. En concreto, es necesario comprobar tipos cuando:

- Se realiza una operación de asignación entre referencias.

```
assign <refDestino>, <refFuente>;
```

- Se invoca un método donde la lista de parámetros es no vacía o el método devuelve un resultado.

```
obj.metodo(valor);
...
method metodo(arg:clasearg) {
   instrucción;
   ...
}
```

Para la asignación de referencias mostrada, es necesario asegurar que la clase de la referencia `refDestino` y la de la instancia apuntada por `refFuente` son compatibles.

Así mismo, en la operación de invocación, la clase de la instancia `valor` debe ser compatible con la clase `clasearg`, ya que de otra manera, no sería posible la invocación del método.

La "compatibilidad" mencionada puede establecerse a diferentes niveles, tal y como ocurre en diferentes lenguajes de programación orientados a objetos. En el caso de la asignación de referencias, por ejemplo, se permitiría la operación en el caso de que la clase de `refDestino` fuese la misma o una superclase de `refFuente` o, en el caso más general, que la clase de `refDestino` fuese la misma o una superclase de la clase de la instancia apuntada por `refFuente`.

Realizada la comprobación de tipos, si tiene éxito, la operación se llevará a cabo; en caso contrario, se elevará una excepción.

En un sistema integral en el que no exista distribución (existe sólo una máquina abstracta), la comprobación de tipos se simplifica notablemente, dado que toda la información acerca de referencias, clases e instancias está disponible en las áreas correspondientes de la máquina abstracta. La comprobación de tipos será realizada en su totalidad por la máquina abstracta, que únicamente tendrá que recorrer y consultar sus estructuras de datos internas (áreas de clases, instancias, etc.).

En el momento en que el sistema integral se convierte en un sistema integral distribuido, la **comprobación de tipos se convierte en un problema importante**. No se puede suponer que toda la información mencionada esté disponible en el nodo en el que se está realizando la comprobación, sino más bien, hay que ponerse en el caso peor: la información sobre las referencias sí está disponible de manera local, pero no necesariamente lo estará toda la información que permita reconstruir la parte necesaria del grafo de clases que permite realizar la comprobación de tipos.

La comprobación de tipos, por tanto, se tiene que convertir en una **comprobación de tipos distribuida**. Dado que las máquinas abstractas son ajenas a la existencia de la distribución, el sistema operativo extenderá reflectivamente la máquina abstracta para la comprobación de tipos, con el fin de poder superar fronteras entre diferentes máquinas del sistema integral.

En esencia, la comprobación de tipos distribuida no se diferencia de la comprobación de tipos en un entorno centralizado, dado que lo que se busca es una respuesta del tipo SI/NO para permitir la realización de una operación. La "única" tarea adicional de la comprobación distribuida será ir a buscar la información de los tipos allá donde se encuentre, es decir, al nodo del sistema integral que contiene la información necesaria para permitir continuar el algoritmo de comprobación. El problema de esta aproximación tal cual es su coste, dado que, en el caso peor, serían necesarios varios mensajes a través de la red para una simple comprobación.

11.5.1 Optimizaciones para la comprobación de tipos

El coste de las operaciones de comprobación de tipos puede ser minimizado consiguiendo que toda la información que interviene se encuentre disponible de manera local en el máximo número de ocasiones.

Una primera mejora consiste en que la clase sombra de cada objeto no se limite a contener el identificador de la clase a la que pertenece, sino que almacenará el conjunto de identificadores correspondientes a todas las superclases de su clase, adecuadamente organizados. Dado que se guardan únicamente los identificadores de las clases, el espacio necesario dentro de la clase sombra no es relevante.

Una segunda mejora consiste en extender el concepto de replicación que se describe en profundidad en el capítulo 13 para permitir la replicación de las clases. De esta forma, la descripción de ciertas (potencialmente, todas) clases estaría disponible en varios (potencialmente, todos) nodos, de manera que se conseguiría de nuevo el objetivo de maximizar aquellas operaciones de comprobación de tipos que podrían ser resueltas de manera local.

11.6 Interacción con la red de comunicaciones

Con el fin de ocultar las particularidades de la red (o redes) de comunicaciones que permite la interacción inter-nodos, es preciso que el sistema operativo disponga de una serie de objetos especializados en el envío y recepción de mensajes a través de la red. Los objetos **comunicadores** serán los encargados de dicha tarea.

La misión de estos objetos es muy simple, ya que únicamente se delega en ellos la ocultación de las características concretas de la red de comunicaciones. La mayor parte de la información que tienen que enviar habrá sido construida por otros objetos del sistema operativo y los comunicadores serán únicamente responsables de hacerla llegar a su destino. Así, por ejemplo, en el caso de que se deseen introducir procesos de cifrado para la información que se transmite, dicho proceso se realizará siempre antes de proporcionar los datos definitivos a transmitir al objeto de comunicaciones.

```
Class Comunicador
Isa object
Methods
   SendMessage(node, string);
Endclass
```

El primer parámetro identificará la máquina de destino, y el segundo, el mensaje a enviar. En una máquina abstracta podrán existir varios objetos comunicadores, cada uno de los cuales hablará un protocolo de red diferente, de manera similar a como ocurre en Spring con los representantes de red [MGH+94].

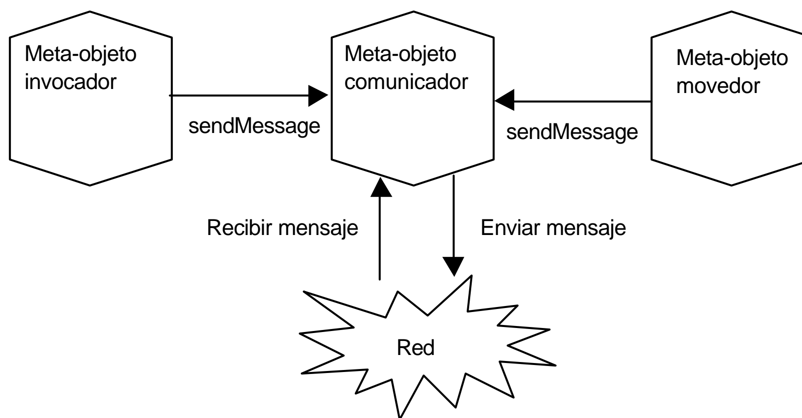


Figura 11.12. Papel del objeto comunicador.

Todo objeto comunicador tiene activo indefinidamente un **hilo de ejecución encargado de escuchar la red**, con el fin de detectar la llegada de mensajes de otros nodos. Ante la llegada de un nuevo mensaje, el objeto comunicador se lo pasará a un objeto del sistema operativo que sabrá qué es lo que tiene que hacer con él. Los mensajes, por tanto, tendrán que construirse de tal forma que el objeto del sistema operativo que los recibe del comunicador pueda conocer la razón de su envío.

11.7 Resumen

Aunque se han identificado la invocación remota y la migración de objetos como las operaciones que caracterizan un sistema operativo distribuido orientado a objetos, no son menos importantes otras operaciones que tendrán que ser realizadas por el sistema operativo debido a la incapacidad de una máquina abstracta de gestionar objetos que no estén ubicados en ella.

La asignación de identificadores y nombres simbólicos a los objetos no es algo nuevo en un sistema operativo como el que se describe en esta Tesis. Todos los objetos tienen que ser mutuamente distinguibles a través de su identificador, con la particularidad de que dicha distinción debe ser conseguida para la globalidad del sistema integral distribuido. Por su parte, los nombres simbólicos son utilizados para facilitar la identificación de aquellos objetos que desean ponerse a disposición del resto. El sistema operativo se encargará, a través del servicio de nombrado, de convertir nombres simbólicos en identificadores.

La localización de objetos es fundamental en un sistema distribuido. Se encarga de obtener información sobre la ubicación de un objeto, dado su identificador. La información de ubicación solicitada puede reflejar con menor o mayor fidelidad la situación real del objeto, obteniéndose así dos semánticas de localización: débil y fuerte, respectivamente. Cada una de ellas será utilizada en diferentes operaciones relacionadas con la distribución de objetos.

La creación y eliminación de objetos en un sistema integral distribuido no es trivial. Debido a la posible dispersión de las clases y los objetos, el sistema operativo debe colaborar con las máquinas abstractas para ir a buscar la información necesaria allá donde se encuentre. El usuario no será consciente de dicha necesidad.

La comprobación de tipos para determinadas operaciones sufre de un problema similar al anterior. Es necesario que el sistema operativo colabore con las máquinas abstractas para recorrer la jerarquía de clases dispersas en el sistema integral distribuido.

Finalmente, un sistema distribuido no sería posible sin una facilidad de comunicación entre sus diferentes nodos. Las comunicaciones quedan ocultas por objetos comunicador, que se encargan de enviar y recibir mensajes a través de la red. Otros objetos del sistema operativo harán uso de estos objetos para completar operaciones de localización, creación y eliminación distribuida, etc.

CAPÍTULO 12 INVOCACIÓN REMOTA Y MIGRACIÓN DE OBJETOS

12.1 Introducción

En el capítulo anterior se describieron un conjunto de mecanismos básicos de soporte de objetos que necesitaban ser sobrescritos por el sistema operativo para un sistema integral distribuido.

En este capítulo se describen las dos facilidades que desde los primeros capítulos de la Tesis se han presentado como imprescindibles para que un sistema operativo orientado a objetos sea también distribuido: la invocación remota y la migración de objetos.

El diseño de ambas facilidades se apoya, de manera fundamental, en la arquitectura reflectiva de la máquina abstracta, haciendo uso de los meta-objetos presentados en el capítulo 9 y otro conjunto adicional de meta-objetos que son definidos y descritos en este capítulo. La descripción de la forma en que se comportan y colaboran los meta-objetos citados permite completar la visión de Agra, el sistema básico de distribución de objetos para el sistema integral.

Los mecanismos que implementan las facilidades mencionadas vienen acompañados de un conjunto de políticas, que determinan cómo han de tomarse determinadas decisiones. Dado que se ha conseguido la separación de políticas y mecanismos, se ha optado por describir una política concreta para cada posibilidad. De todas formas, en varios casos se apunta qué otras políticas podrían ser utilizadas para conseguir determinados objetivos.

12.2 Invocación de métodos

Se ha planteado como uno de los objetivos principales del **sistema de distribución para el sistema integral que sea transparente**, es decir, que el usuario/programador que tenga que trabajar/desarrollar en él no necesite saber, para realizar su tarea, si el sistema integral soporta actualmente la distribución o no (es decir, si el sistema integral está compuesto por 1 o más nodos).

En el caso de la invocación de métodos, la transparencia viene dada por el hecho de que el usuario/programador no tenga que saber si el objeto al que desea invocar está ubicado en el mismo o en otro nodo del sistema integral o incluso si se encuentra en el almacenamiento de persistencia. Es necesario proporcionar al usuario la ilusión de que los objetos que desea invocar se encuentran siempre disponibles, independientemente de su ubicación real. El sistema operativo, cooperando con la máquina abstracta, conseguirá proporcionar la mencionada transparencia/ilusión en este ámbito.

Ya en el capítulo 9 se apuntó la forma de introducir la invocación remota en el sistema integral, modificando el funcionamiento que presentan por defecto los meta-objetos emisor y receptor del sistema operativo. A continuación se describe en detalle

todo el proceso, tanto para la invocación síncrona como para una posible invocación asíncrona.

12.2.1 Invocación síncrona

La invocación síncrona de un objeto se concreta en la ejecución de una instrucción (por ejemplo, *call*) por parte de la máquina abstracta, que dada su arquitectura reflectiva, cederá el control al sistema operativo para la resolución de la invocación. Dado que la invocación es síncrona, la máquina abstracta bloquea el hilo en el objeto base hasta que finalicen todas las tareas relativas a la invocación.

El meta-objeto emisor del objeto cliente[13] será el que obtenga el control de la máquina abstracta, procediendo a realizar su parte de la operación de invocación, que consiste básicamente en la localización del objeto servidor[14], la preparación y envío del mensaje que contiene los datos de la invocación y la obtención del mensaje con el resultado de la invocación (que podrá ser una excepción).

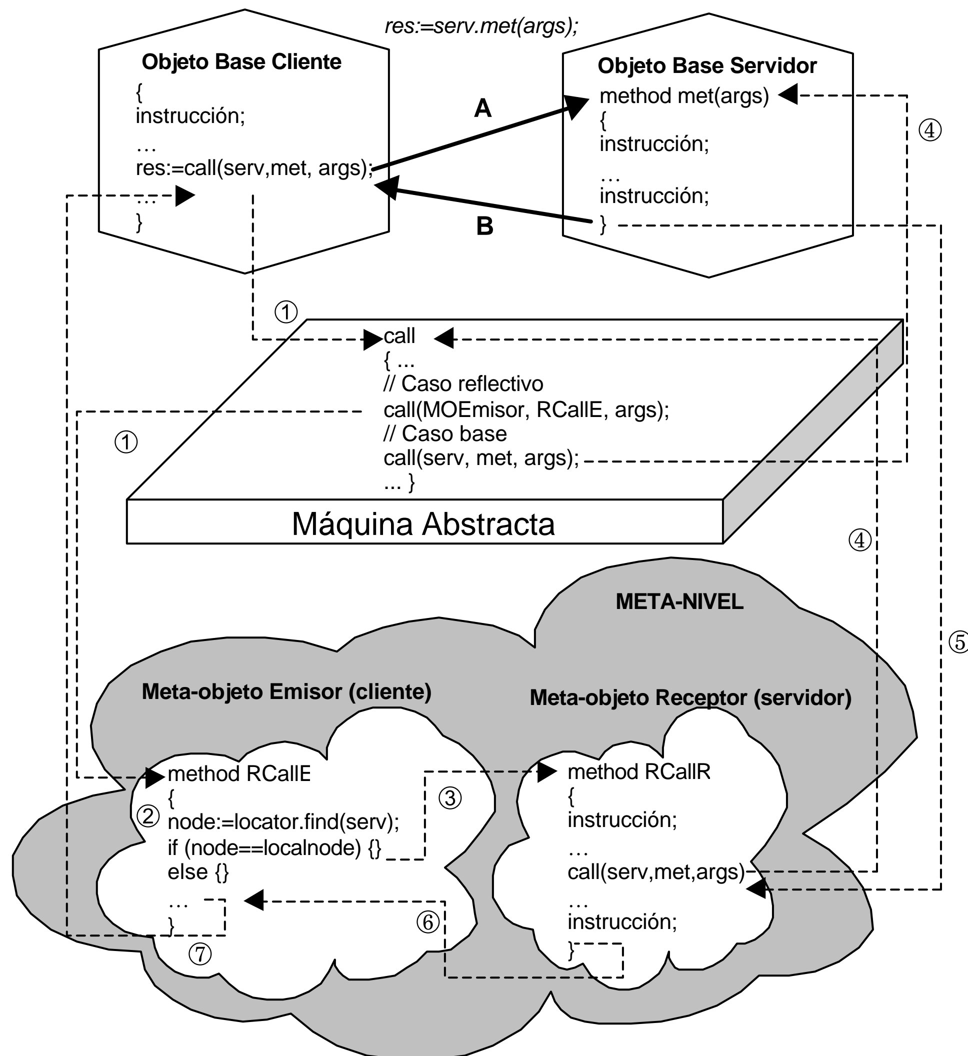


Figura 12.1. Proceso de invocación de un método.

[13] A partir de ahora se utilizarán indistintamente los términos objeto cliente y objeto invocador
[14] A partir de ahora se utilizarán indistintamente los términos objeto servidor y objeto invocado

El meta-objeto receptor del objeto servidor recibirá en algún momento el mensaje enviado por el meta-objeto emisor, cediendo el control al nivel base para que se realice la ejecución efectiva del método invocado, y preparando el mensaje que deberá ser retornado al meta-objeto emisor, con lo que concluirá su trabajo.

En la figura 12.1 se muestra gráficamente el proceso de invocación. El objeto base cliente invoca al objeto base servidor con la instrucción `res:= call(serv, met, args)`. La referencia `res` contendrá el resultado de la invocación. La referencia `serv` apunta al objeto base servidor. La referencia `met` indica el método a ejecutar en el objeto servidor. Finalmente, la referencia `args` representa la lista de parámetros a pasar al método indicado.

Lo que el programador del nivel base desea ver del proceso de invocación está representado por las flechas etiquetadas con **A** y **B**: la invocación de un método del objeto servidor produce la ejecución de éste, y la devolución del control al objeto cliente se producirá cuando finalice la misma.

Pero lo que realmente ocurre está identificado por las flechas etiquetadas con números. En primer lugar, la ejecución de la instrucción de invocación de un método en el nivel base produce la transferencia de control al meta-nivel, donde toma el control el meta-objeto emisor del objeto cliente ①.

A continuación, es necesario **comprobar si la invocación realizada por el objeto cliente va a tener carácter local o remoto** ②. En cualquiera de los dos casos, sería necesario enviar un mensaje con los parámetros de la invocación (`met`, `args`) al meta-objeto receptor del objeto servidor ③ para completar la parte servidora de la invocación. La diferencia básica en este paso entre una invocación local y una remota es que la invocación remota necesitará que el mensaje sea enviado a través de la red de comunicaciones que une los diferentes nodos del sistema integral distribuido.

Una vez que el meta-objeto receptor haya recibido el mensaje, cederá el control a su objeto base, con el fin de que pueda tener lugar la invocación efectiva del método (`met`) invocado ④. Finalizada la ejecución del método, sólo queda devolver el control con el resultado de la ejecución ⑤, ⑥ y ⑦. De nuevo, el mensaje que contiene el resultado de la ejecución del método del objeto servidor puede necesitar ser enviado a través de la red de comunicaciones si los objetos cliente y servidor residen en diferentes nodos ⑥.

La descripción realizada de la invocación de métodos permite conocer, a grandes rasgos, qué **acciones se realizan en el meta-nivel para conseguir la invocación transparente de objetos**. En realidad, las operaciones a realizar en el meta-nivel involucran un conjunto mayor de operaciones y meta-objetos, que se describen en los apartados siguientes en todo su detalle y siguiendo un estricto orden secuencial.

12.2.1.1 Preparación de la invocación en el meta-objeto emisor

Lo primera tarea a realizar por el meta-objeto emisor en el momento en que toma el control es **localizar el objeto servidor y fijarlo**. Aunque, en general, la localización fuerte es la más conveniente, dado que proporciona información actualizada sobre la ubicación del objeto, tiene el problema de ser más costosa. Por lo tanto, **se empezará utilizando de manera optimista una localización débil**, en el nodo del objeto cliente. En el nodo del objeto servidor, se utilizará una localización fuerte, que en la mayor parte de los casos se resolverá de manera local. La localización débil utilizada en el nodo cliente es suficiente en la mayor parte de los casos, donde el objeto servidor no se ha movido. Una ventaja es que los objetos que no se mueven frecuentemente pueden ser

invocados de manera eficiente, es decir, el rendimiento de la invocación remota no se degrada por la existencia de la migración.

Para localizar el objeto, el meta-objeto emisor invoca el objeto localizador del sistema operativo, pasándole una copia de la referencia del objeto servidor (`serv`) y la opción de localización para fijarlo. Dado que no se dice lo contrario, se realizará una localización débil del objeto (únicamente en la cache de direcciones de objetos no locales) una vez que se ha comprobado que el objeto no reside en la máquina local, es decir, no está en su área de instancias.

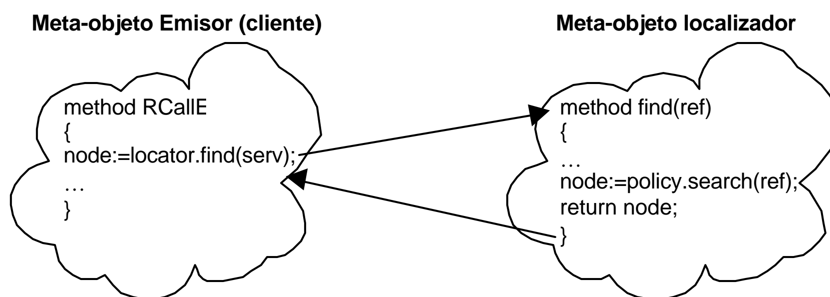


Figura 12.2. Relación entre los meta-objetos emisor y localizador.

En los apartados siguientes se describen las posibles situaciones que se pueden producir en función de la información obtenida en la cache de direcciones de objetos no locales acerca del objeto servidor.

El objeto servidor está en la máquina local

El objeto localizador habrá consultado el área de instancias local y habrá comprobado que el objeto servidor se encuentra en ella. El objeto es fijado en la máquina local para que no se pueda mover hasta que el mensaje que contiene la invocación sea recibido por su meta-objeto receptor. En ese momento se liberará su atadura al nodo, de tal forma que pueda ser elegido para migrar aun cuando no se haya completado la operación de invocación.

El objeto servidor está en el almacenamiento de persistencia

Si el objeto localizador ha encontrado el objeto en el almacenamiento de persistencia, se pedirá al sistema de persistencia que lo active en algún nodo del sistema integral, obteniéndose la dirección de dicho nodo como resultado de la activación.

El objeto servidor está en una máquina remota

Si el objeto localizador ha encontrado el objeto en algún otro nodo del sistema integral, devolverá dicha dirección.

El objeto servidor está migrando

Si el objeto servidor está migrando, se encuentra en un estado transitorio hasta que finaliza su movimiento y se reactiva en el nodo de destino. Será necesario retardar el envío del mensaje, dado que no podrá ser atendido en este momento. A partir de este momento, se reintentará la localización del objeto hasta que se obtenga una respuesta satisfactoria para el objeto ya activo en el nodo destino de su migración.

El objeto servidor no existe

Esto quiere decir que no se tiene ninguna información sobre la ubicación del objeto. Será necesario realizar una localización fuerte para obtener algún dato acerca de su

ubicación, dado que no se puede presumir que el objeto no existe en el sistema integral simplemente porque no se disponga de información acerca de la ubicación del mismo.

Preparación y envío del mensaje

Llegados a este punto, el objeto servidor se encuentra activo en algún nodo del sistema integral y habrá que hacerle llegar el mensaje que contiene la información de la invocación. Dependiendo del resultado de la localización, la invocación será local (objetos cliente y servidor residen en el mismo nodo) o remota (residen en nodos distintos).

En el caso de que la **invocación sea local**, el meta-objeto emisor invoca directamente al meta-objeto receptor del objeto servidor, o dicho de otra forma, el meta-objeto emisor envía un mensaje al objeto servidor (`serv`) y dicho mensaje es interceptado por el meta-objeto receptor, que decide qué hacer con el mismo. Lo habitual es que lo procese, cediendo el controla al nivel base para que se ejecute el método realmente invocado por el objeto cliente. Al tratarse de una invocación local, el paso de mensaje es realizado por la propia máquina abstracta.

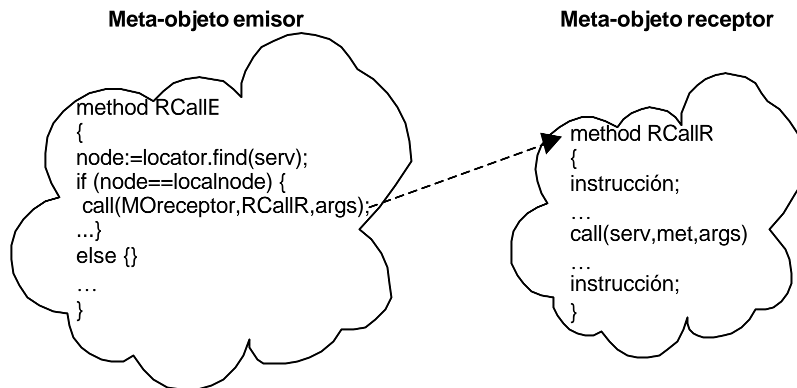


Figura 12.3. Invocación local.

En el caso de que la **invocación sea remota**, el meta-objeto emisor construye el mensaje que contiene los datos de la invocación e invoca al objeto **comunicador** para que haga llegar el mensaje al nodo en el que reside el objeto servidor. Además, dado que se trata de una invocación síncrona, el meta-objeto emisor se queda esperando a que finalice el resto de pasos de la invocación, para devolver a su vez el control al objeto base.

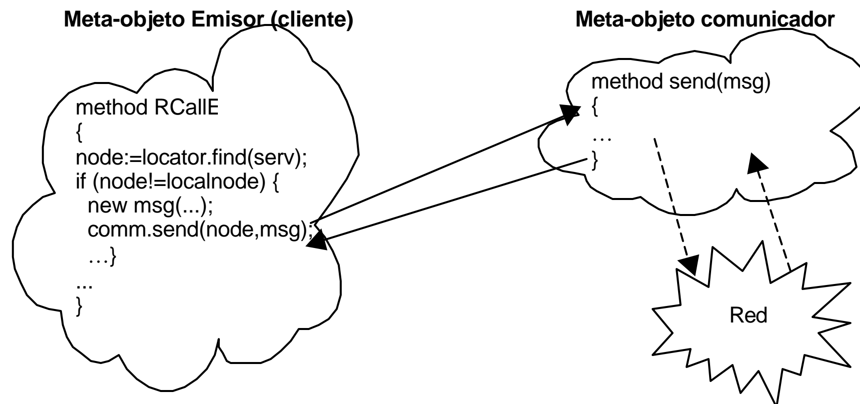


Figura 12.4. El meta-objeto emisor invocado al objeto comunicador para que tenga lugar una invocación remota.

El **mensaje a enviar** en una invocación remota incluirá la información siguiente:

- La **ubicación del objeto cliente**. Esta información de ubicación será utilizada para la devolución del resultado de la invocación al método.
- Una **referencia al objeto que realiza la operación de invocación (cliente)**. La identidad de este objeto es necesaria para el mensaje de retorno de la invocación y para que el objeto servidor, en caso de que así lo desee, pueda incluir la identidad del cliente como parte de los datos a considerar para la ejecución del método. Dado que la identidad del objeto forma parte de cualquier referencia al mismo, el objeto servidor podrá obtenerla sin dificultad.
- La **referencia del objeto servidor** que posee el objeto cliente. Es importante destacar que es necesario pasar la referencia y no la identidad del objeto invocado para que en la máquina abstracta en la que reside el objeto servidor se puedan llevar a cabo las operaciones de control de permisos (protección) pertinentes (ver capítulo 14).
- Una lista de referencias a los **objetos parámetros formales** de la invocación, dado que el paso de parámetros es siempre por referencia. De nuevo aparece la necesidad de utilizar referencias a los objetos, por el problema de la protección de los objetos.

Opcionalmente, se puede introducir en esta fase un proceso de **cifrado del mensaje**, de tal forma que circule de manera segura por la red de comunicaciones. El cifrado del mensaje se realiza antes de su entrega al objeto comunicador, de tal forma que este se encarga únicamente de hacerlo llegar al nodo destino. Los procesos de cifrado y descifrado serían realizados por **meta-objetos cifrador**, presentes en todos los nodos del sistema integral distribuido y que actuarían de manera coordinada por el sistema operativo.

12.2.1.2 Recepción del mensaje por el meta-objeto receptor

El meta-objeto receptor tomará el control como resultado de una invocación a su objeto base asociado. Ahora bien, el meta-objeto que lo invoca dependerá del carácter de la invocación: local o remota.

Si la **invocación es local**, el meta-objeto emisor asociado al objeto base cliente será el que lo habrá invocado. En el caso de una invocación remota, no es posible que sea

así, dado que las máquinas abstractas sólo conocen los objetos almacenados en sus áreas de instancias. Es necesario, entonces, que algún otro objeto juegue el papel del objeto invocador en la máquina remota.

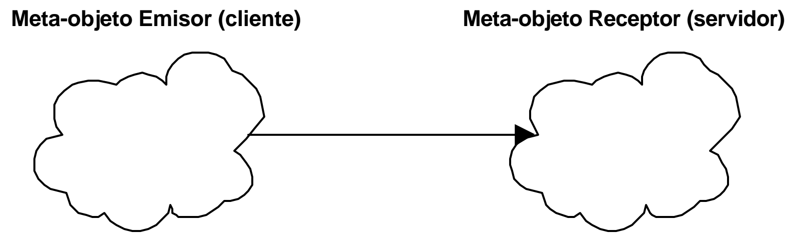


Figura 12.5. Invocación local.

En una **invocación remota** el mensaje sale de la máquina local a través del objeto comunicador. En la máquina remota dicho mensaje es recibido a su vez por un objeto comunicador que se encuentra permanentemente a la escucha. Una vez recibido el mensaje, el objeto comunicador crea un objeto **trabajador**, al que le pasa el mensaje recibido, para que lo procese.

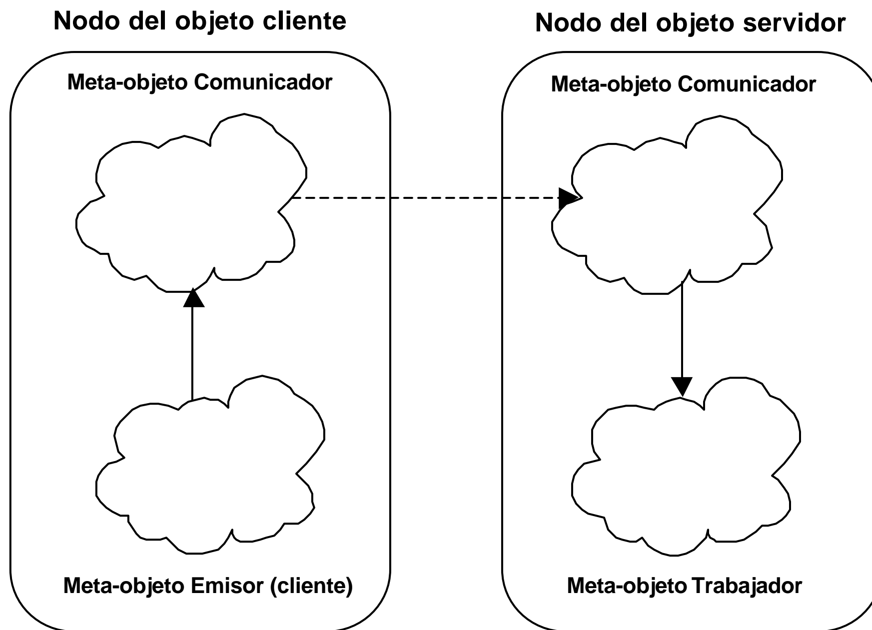


Figura 12.6. Invocación remota.

El objeto trabajador es un objeto del sistema operativo que conoce el formato de los mensajes que intercambian las diferentes máquinas abstractas, y, por tanto, conoce la naturaleza de todos y cada uno de ellos y el tratamiento que tienen que recibir en la máquina local. En este caso, la información contenida en el mensaje es la necesaria para que pueda realizar la invocación del objeto servidor.

Lo primero que tiene que hacer el objeto trabajador es descifrar el mensaje, en el caso de que estuviera cifrado, con la ayuda de un meta-objeto cifrador para tener acceso a su contenido. La información que en un principio va a necesitar del mensaje es la identidad del objeto servidor.

A continuación, el objeto trabajador debe **comprobar que realmente el objeto servidor se encuentra en ese nodo**. Para ello, invoca al objeto localizador de la máquina abstracta en la que reside para que busque el objeto servidor utilizando una **política de localización fuerte**.

La **primera operación de localización**, realizada en el nodo local, se llevó a cabo con una política de localización débil, ya que se supone que, en general, los objetos se mueven con menos frecuencia que con la que son invocados, por lo que hay muchas posibilidades de que la pista que se tenga sobre su ubicación sea buena.

La **repetición de la operación de localización** del objeto no es muy costosa, dado que se espera que en la mayor parte de las ocasiones se encuentre el objeto en la propia máquina. Si el objeto no está presente en la máquina, la ubicación del objeto no se corresponde con la información que se tenía al respecto, por lo que se realiza una localización fuerte, que dará como resultado la verdadera ubicación del objeto. Una vez obtenida la verdadera y definitiva ubicación del objeto, se reenvía el mensaje con los datos de la invocación a dicho nodo, donde se procede de nuevo con la recepción del mensaje por parte del objeto comunicador. El objeto trabajador creado en el primer nodo visitado por el mensaje, termina su ejecución. Si realizada la localización fuerte, no se encuentra el objeto (no existe en el sistema integral), se devolverá una excepción al objeto cliente indicándole tal error.

Una vez que el objeto trabajador está seguro de que el objeto servidor está ubicado en su mismo nodo, lo fija en él. La invocación ya tendrá carácter local, por lo que se pasa al caso más simple de invocación a un objeto. El objeto trabajador enviará un mensaje al meta-objeto receptor del objeto servidor, envío que será realizado por la máquina abstracta.

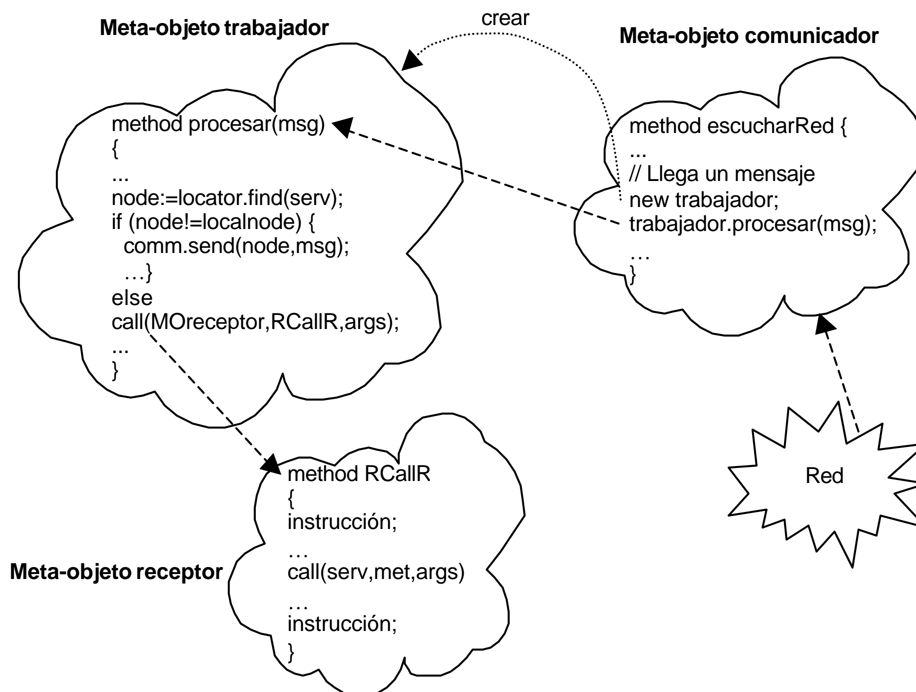


Figura 12.7. Creación del objeto trabajador para completar la invocación.

Finalmente, y dado que se espera que las referencias que forman la lista de parámetros de la invocación sean utilizadas por el objeto servidor, el objeto trabajador puede insertarlas en la cache de direcciones de objetos no locales. De esta manera, tan

pronto como el objeto servidor invoque alguno de los métodos de los objetos apuntados por las referencias mencionadas, la operación de localización de los mismos ser verá muy acelerado.

12.2.1.3 Invocación efectiva del método en el objeto base

Dado que el objeto servidor permanecía fijado al nodo desde el momento en que el objeto localizador lo localizó, y dado que el proceso de invocación está en la fase de interacción entre el meta-objeto receptor y su objeto base (el objeto servidor), el meta-objeto receptor liberará el objeto, de forma que recupere la libertad para moverse si así lo determina el sistema operativo.

Una vez que el meta-objeto receptor ha liberado de su atadura a su objeto base asociado, debe realizar un conjunto final de acciones antes de ceder el control al nivel base para que se pueda realizar la ejecución efectiva del método.

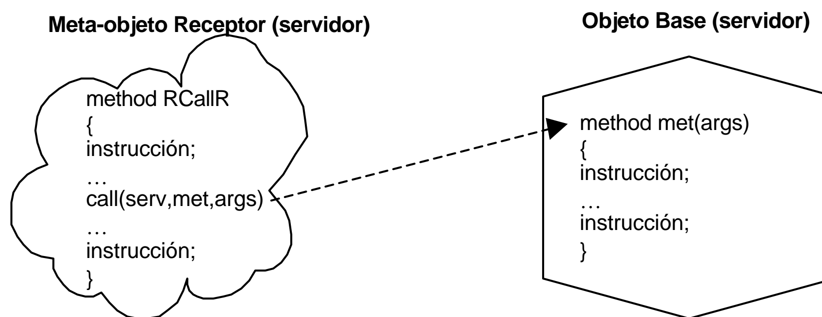


Figura 12.8. Invocación efectiva del método invocado en el objeto servidor.

El conjunto final de acciones mencionado viene determinado por la necesidad de **comprobar si el objeto servidor dispone de la definición del método** invocado. Dado que es posible que un objeto resida en un nodo en el que no está definida la clase a la que pertenece (o alguna de sus superclases), no se puede asegurar que la definición del método invocado esté disponible localmente.

En el caso de que la definición del método esté disponible (el método invocado está definido en la clase del objeto o en alguna de sus superclases disponibles localmente o bien existe una copia del mismo en su clase sombra) se puede ceder definitivamente el control al nivel base para la ejecución efectiva del método.

En el caso de que no esté disponible la definición del método, será necesario **obtener una copia del mismo bajo demanda**, de una manera similar al funcionamiento de la paginación por demanda en un sistema de gestión de memoria virtual [Sta98].

Dado que se conoce la identidad de la clase a la que pertenece el objeto (toda instancia guarda una referencia a su clase) y cada clase guarda información acerca de sus superclases, se inicia una operación de búsqueda hacia arriba (de descendientes a ascendientes) en el grafo de herencia para buscar el método invocado. De la misma forma que ocurría con la operación de comprobación de tipos, la búsqueda del método puede involucrar a varios nodos del sistema integral.

Si la operación de búsqueda acaba con éxito, se envía una copia de la definición del método y cierta información para la comprobación de permisos al nodo en el que reside el objeto servidor. La copia del método y la información de comprobación de permisos se insertan en la clase sombra del objeto servidor.

En el caso de que la búsqueda termine en fracaso, la invocación del método terminará con una excepción del tipo METHOD_NOT_FOUND que se hará llegar al objeto cliente.

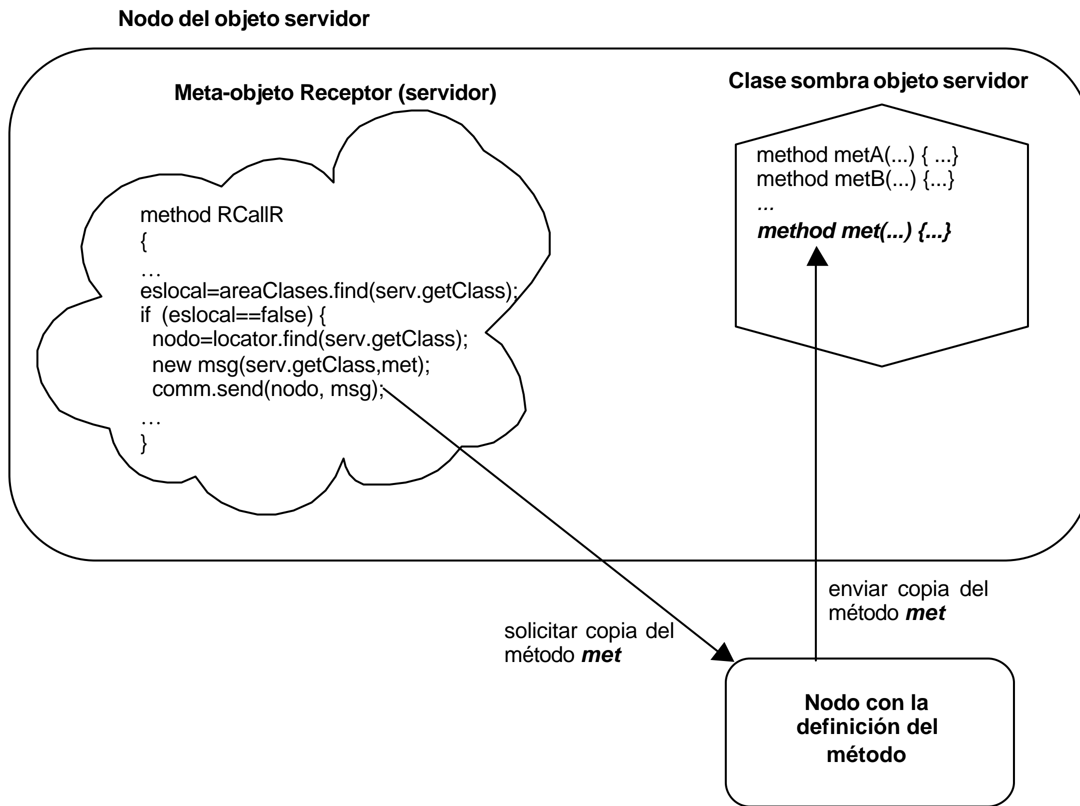


Figura 12.9. Obtención de la definición de un método bajo demanda.

En el caso de que la búsqueda termine con éxito, es necesario realizar una última operación antes de tener lugar la ejecución efectiva del método: la comprobación de tipos de las referencias que forman la lista de los parámetros de la invocación.

12.2.1.4 Devolución del resultado de la invocación

Tras la ejecución efectiva del método invocado, el resultado de la misma puede ser:

- **Vacío**. La definición del método no especificaba resultado alguno.
- Una **referencia a un objeto**.
- Una **excepción**.

En cualquiera de los tres casos es necesario construir un mensaje de retorno, que tendrá como origen el meta-objeto receptor del objeto servidor y como destino el meta-objeto emisor del objeto cliente.

Como se describió anteriormente, la invocación que ha tenido lugar tiene, en cierto modo, un componente local, dado que ha sido realizada por el objeto trabajador creado al efecto. Este objeto será el que reciba, en primera instancia, el resultado de la invocación y el responsable de hacer llegar dicho resultado al objeto cliente. En primer lugar, construirá el mensaje que contiene toda la información del resultado del método, cifrándolo si así se ha dispuesto. En segundo lugar, proporcionará el mensaje al objeto comunicador, que la hará llegar al nodo en el que reside el objeto cliente. Una vez que

el mensaje llega al nodo en el que residía el objeto cliente, se realiza una localización fuerte para determinar si el objeto al que hay que entregar el resultado está en ese nodo o es necesario reenviar dicho mensaje al nodo en el que efectivamente se encuentra.

12.2.1.5 Recepción del resultado por el meta-objeto emisor

El objeto comunicador del nodo cliente recibirá el mensaje que contiene la respuesta, mensaje que hará llegar al meta-objeto emisor del objeto cliente, que estaba esperando a recibirlo. El meta-objeto emisor estudiará el contenido del mensaje y se lo hará llegar a la máquina abstracta de manera reflectiva, de tal forma que esta realice las acciones necesarias para entregar el resultado al objeto base cliente:

- Si el resultado de la invocación es una referencia a un objeto, dicha referencia se copiará en la referencia que el objeto cliente hubiera dispuesto. En caso de que sea necesaria una conversión de tipos y la información necesaria no esté disponible localmente, la máquina abstracta solicitará la ayuda del sistema operativo para realizarla.
- Si el resultado de la invocación es una excepción, la máquina abstracta la retransmitirá al objeto base cliente. Realmente, el meta-objeto emisor del objeto cliente se encarga de generar una excepción del mismo tipo que la recibida en el mensaje procedente del meta-objeto receptor del objeto servidor.
- Si no hay referencia resultado ni excepción, la máquina cede inmediatamente el control al objeto base cliente.

En cualquiera de los casos anteriores, el objeto base cliente pasa a tener el control para ejecutar la siguiente instrucción o manejar una excepción pendiente.

Como operación adicional ante la recepción del resultado, el meta-objeto emisor del objeto cliente **actualizará la información de ubicación del objeto** servidor en la cache de direcciones de objeto no locales. De esta forma se consigue que las futuras operaciones de localización del objeto próximas en el tiempo puedan ser satisfechas con éxito con una localización débil.

12.2.2 Invocación asíncrona

La invocación asíncrona se realiza con la instrucción *send*. El objeto del nivel base podrá continuar su ejecución cuando se haya asegurado de que se han llevado a cabo todas las acciones necesarias para la ejecución del método (no hace falta que el método se haya ejecutado ya).

En la figura siguiente se muestra el proceso necesario para una invocación asíncrona. Como ya se comentó en la invocación síncrona, lo que el programador desea ver del proceso de invocación está representado por las flechas etiquetadas con **A** y **B**: la invocación del método del objeto servidor produce la ejecución de éste, y la devolución del control se producirá tan pronto como se haya enviado el mensaje.

Dado que la invocación es asíncrona, el objeto cliente deberá poder continuar su ejecución tan pronto como es seguro que el objeto servidor ha recibido su invocación. Cuando el objeto cliente invoca el método del objeto receptor ①, se produce una transferencia de control al meta-nivel, donde toma el control el meta-objeto emisor ②. El meta-objeto emisor comprueba si la invocación tiene carácter local o remoto, y procede en consecuencia, enviando el mensaje al meta-objeto receptor del objeto servidor ③. Dado que la invocación es asíncrona, el meta-objeto receptor ejecuta la instrucción `send` ④, que además de poner en ejecución el método del objeto base,

devuelve el control inmediatamente al meta-objeto emisor del objeto cliente. El objeto cliente obtendrá a su vez el control y podrá continuar su ejecución, sin necesidad de esperar que el método invocado haya finalizado.

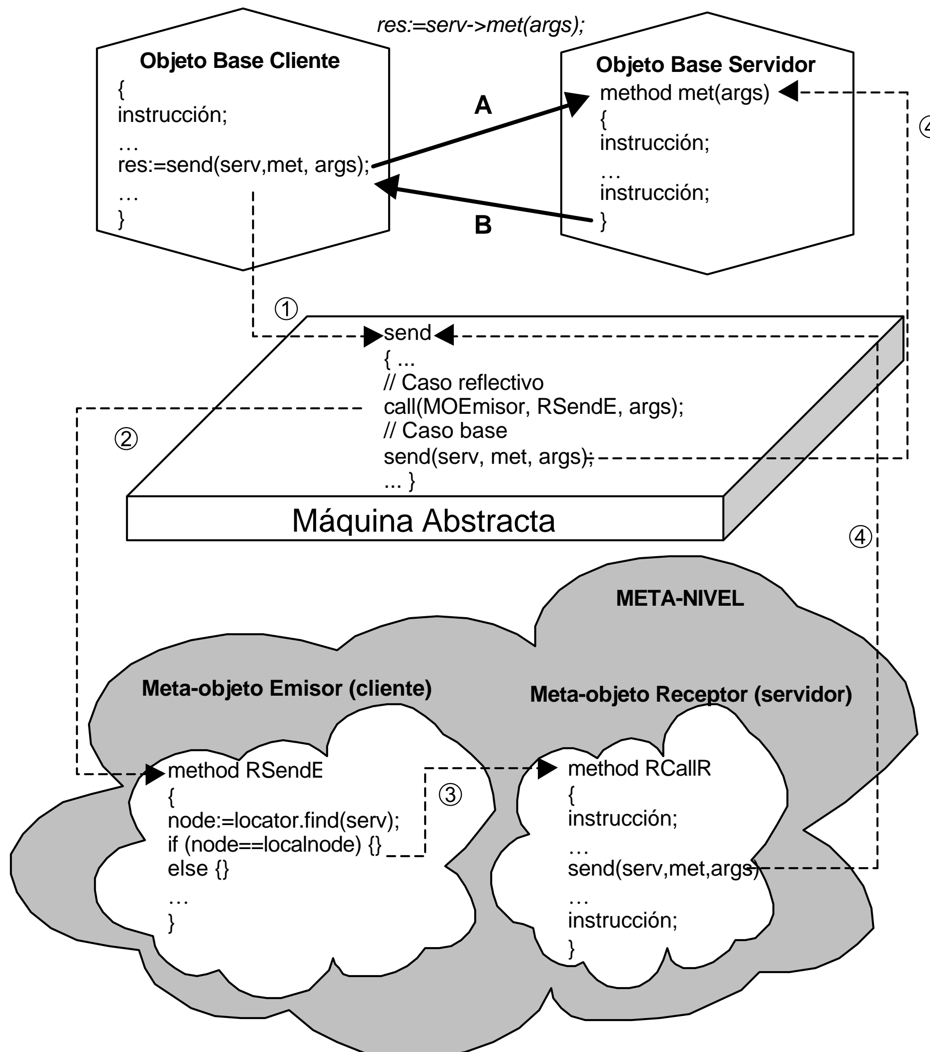


Figura 12.10. Proceso de invocación asíncrona.

12.3 Propagación de excepciones

Como se describió en el capítulo 3, las excepciones son un concepto importante de la definición del modelo de objetos del sistema integral. La introducción de la distribución en el sistema complica la gestión de excepciones en tiempo de ejecución, dado que el objeto que produce la excepción y el manejador de la excepción pueden no residir en la misma máquina abstracta.

¿Dónde se puede encontrar el manejador de la excepción? Depende de la naturaleza, local o remota, de las invocaciones a métodos encadenadas hasta la invocación en curso. Si el método en el que se lanza la excepción no tiene definido un manejador para la misma, deberá propagar a su objeto llamador la excepción sin manejar, de tal modo que en el método de este último objeto desde el que se hizo la invocación se buscará de nuevo el manejador, y así sucesivamente.

Hay que tener presente que la operación de propagación de excepciones debe ser capaz de superar fronteras entre máquinas abstractas distintas, y que a la hora de buscar un objeto para enviarle la excepción, quizá haya sido movido hacia otra máquina. En este caso entraría en juego el servicio de localización, que buscaría el objeto en el sistema integral, de la misma manera en que realiza las búsquedas para la invocación de métodos.

Realmente, el proceso de **propagación remota de excepciones se solapa con el proceso de devolución de resultados** de las invocaciones remotas. Dado que para una invocación sólo puede ocurrir una de las dos cosas, el mensaje devuelto desde el objeto servidor al objeto cliente contendrá información sobre la situación que realmente se produjo.

12.4 Migración de objetos

12.4.1 Problemas que presenta la migración de objetos

La migración de objetos introduce una serie de problemas en su gestión. Todos ellos han de ser solucionados por la combinación de mecanismos y políticas que implementan la migración y que se describirán después en mayor profundidad:

- Determinar el estado del objeto. El estado de un objeto viene determinado por el valor de sus atributos y el estado de su computación interna. Es preciso poder "congelar" el objeto desde que se decide moverlo hasta que ya se encuentra en su destino, con el fin de que pueda reanudar sus tareas como si nada hubiera ocurrido.
- El coste asociado a la migración de un objeto puede sobrepasar los beneficios del movimiento. Se trata, en general, de evaluar si es rentable mover un objeto determinado en un momento concreto. La utilización de políticas adecuadas permitirá decidir con máxima precisión.
- Las peticiones de invocación enviadas mientras se está moviendo el objeto han de ser atendidas de manera transparente. En el momento en que el objeto haya finalizado la migración, deberá ser capaz de atender tanto las peticiones que ya estaban en curso, como las que llegaron cuando se estaba moviendo.
- Los objetos no deben estar moviéndose continuamente. La utilización de políticas de migración correctas evitará que los objetos se muevan indefinidamente, dándoles la oportunidad de ejecutar sus métodos, sirviendo así las peticiones que al respecto hagan otros objetos.

12.4.2 Decisiones de diseño

Es necesario tomar varias decisiones de diseño a la hora de desarrollar la facilidad de migración de objetos para el sistema integral. Los aspectos más importantes a tener en cuenta son los siguientes, alguno de los cuales ha sido tratado ya de manera genérica (ver capítulos 2 y 3), pero que se tratarán ahora específicamente para la migración de objetos: separación de políticas y mecanismos, independencia entre mecanismos, transparencia y eficiencia.

12.4.2.1 Separación de políticas y mecanismos

Como ya se ha mencionado en otras ocasiones, la separación de políticas y mecanismos es muy importante en el diseño de sistemas operativos [Gos91]. En lo que

se refiere a la migración de objetos, las políticas tratan de definir cuándo migrar qué objetos y adónde. Realmente se trata de resolver el problema de la asignación dinámica de objetos a máquinas abstractas (procesadores, en el caso más general).

Por otro lado, un mecanismo de migración proporciona las facilidades para incorporar la política, es decir, recoge las estadísticas necesarias para tomar las decisiones marcadas por las políticas y realiza la migración real del objeto. La separación de políticas y mecanismos permite una implementación eficiente de estos últimos, que no necesitará ser cambiada aunque sí cambie la forma en que se toman las decisiones de migración (políticas).

12.4.2.2 Independencia entre mecanismos

Los mecanismos de migración de objetos, en particular los de obtención de estadísticas y la migración en sí, deben ser diseñados de manera que no interfieran con otros mecanismos del sistema. De nuevo se garantiza que los mecanismos pueden ser fácilmente modificables o reemplazables. En particular, es muy importante que la migración de objetos no interfiera con la invocación de métodos. Por ejemplo, un objeto de usuario no debe detectar problema alguno (transparencia) cuando invoca un método de un objeto que está migrando.

12.4.2.3 Transparencia

La migración debe ser transparente para los objetos, es decir, ni el propio objeto migrado ni los objetos con los que interactúa deben verse afectados por la migración. Para conseguir una migración transparente, la facilidad de migración debe satisfacer los siguientes requisitos:

- Los objetos no tienen que incluir código adicional para gestionar excepciones derivadas de la migración.
- Los objetos referenciados por un objeto migrado tienen que seguir siendo identificados de manera correcta.
- Las operaciones realizadas por la facilidad de migración no son visibles para los objetos de usuario.

12.4.2.4 Eficiencia

La eficiencia es uno de los aspectos de diseño principales de la migración en particular y de cualquier sistema en general. La eficiencia se puede expresar en unidades de tiempo, dado que es uno de los recursos críticos de la ejecución. A la hora de migrar objetos, es necesario tener en cuenta todos los factores siguientes que afectan al rendimiento de la operación:

- El tiempo necesario para seleccionar un objeto para ser migrado.
- El tiempo necesario para determinar el nodo de destino.
- El tiempo necesario para activar el objeto en el nodo destino y eliminar su rastro en el origen.
- El tiempo necesario para el movimiento propiamente dicho.

Todos estos tiempos tienen que ser tenidos en cuenta a la hora de diseñar o concretar la facilidad de migración.

12.4.3 Migración de objetos en el sistema integral

En cualquier momento, un objeto puede ser elegido para ser movido a otro nodo del sistema integral, ya sea porque lo ha solicitado el propio objeto o porque el sistema operativo así lo haya decidido (ver apdo. 10.6.3.1). En cualquier caso, la razón última por la que se mueve un objeto vendrá dada por la política de elección de los objetos a mover, que integrará las respuestas al **cuándo**, **qué** y **adónde** mover.

Cuando un objeto se mueve, es necesario asegurar que la operación de migración (el **cómo**) se realiza de manera atómica, es decir, **la migración de un objeto no puede alterar su correcto funcionamiento**. Todos los hilos de ejecución que estuvieran activos en el momento que se toma la decisión de mover el objeto deberán reanudarse de manera transparente en el momento en que el objeto se activa en el nodo destino. Además, las invocaciones que pudieran llegar al objeto mientras se está moviendo deberán ser tratadas.

De manera elemental, el conjunto de acciones a realizar para mover un objeto es el siguiente: sacar una copia del estado encapsulado del objeto a mover; llevárselo al nodo de destino y reactivar el objeto en dicho nodo, descartando el antiguo. Dado que en determinadas situaciones, y en función de la política de movilidad en funcionamiento, puede ser necesario mover grupos de objetos desde un origen a un destino comunes, lo descrito anteriormente es igualmente válido para cada uno de los objetos individuales de dicho conjunto.

El sistema operativo dispondrá de objetos de tipo **movedor**, que implementarán, con la colaboración de otros objetos, todo el proceso de mover un objeto, dada una referencia al mismo y la dirección de la máquina a la que se desea mover. El objeto movedor será invocado con frecuencia por otros objetos del sistema operativo, como los encargados del **equilibrado de carga**, con el fin de mover los objetos que con tal motivo hayan sido escogidos. Así mismo, el objeto movedor podrá ser invocado por objetos de usuario que deseen tener control sobre la distribución para, por ejemplo, solicitar al sistema operativo la migración de determinados objetos o fijar la ubicación de un objeto en un nodo determinado.

A su vez, el objeto movedor utilizará los servicios de otros objetos de más bajo nivel (objeto comunicador) para permitir la transmisión, a través de las líneas de comunicación disponibles, de toda aquella información necesaria para la migración.

Finalmente, la migración de objetos no sería posible sin el concurso de los meta-objetos que implementan la reflectividad estructural y de comportamiento del sistema integral.

12.4.4 Cómo mover un objeto

La existencia o no de actividad interna en el objeto a mover determina, inicialmente, la dificultad de la operación de migración. Los objetos que no tienen hilos activos son los más fáciles de mover. Básicamente, lo único que se tendrá que mover de una máquina abstracta a otra es el valor de los atributos del objeto.

Si el objeto presenta algún tipo de actividad interna, debido a que alguno de sus métodos ha sido invocado, será necesario añadir el estado de la computación a la información a transmitir para completar la migración del objeto. El código correspondiente a los métodos que tienen hilos activos formará parte también de la información a transmitir. Esto es así porque no se puede suponer que dicho código vaya a estar disponible en el nodo destino y es deseable que el objeto pueda reanudar su

actividad en el nodo destino tan pronto como sea planificado. El código correspondiente al resto de los métodos del objeto no se enviará con la información de migración. Se utiliza esta **política perezosa** para los métodos no activos porque el código tendría que ser transmitido de todas maneras si no estaba presente en el nodo remoto, y se ahorra la transmisión para aquellos métodos que nunca vayan a ser invocados. Por otro lado, el coste asociado a esta transmisión perezosa de los métodos es únicamente un mensaje en el que se solicita su código, coste mínimo si lo comparamos con el coste de transferir el código de un método.

El objeto movedor, una vez que ha recibido la orden (ha sido invocado) de mover un objeto determinado, deberá realizar un conjunto preestablecido de operaciones. Dichas operaciones consisten en la invocación de objetos ya descritos, como los objetos reflejados de la máquina abstracta y los meta-objetos asociados a todo objeto. A continuación se describen, en orden secuencial, el conjunto de operaciones a realizar por el objeto movedor para completar con éxito una operación de migración de un objeto.

12.4.4.1 Obtener una copia del estado encapsulado del objeto

Para obtener la copia del estado encapsulado del objeto, el objeto movedor tiene que llevarlo a un **estado consistente** [BHD+98]. Un estado consistente para un objeto es aquel en el que no existe ningún tipo de actividad interna en el objeto, de manera que, a partir de dicho estado, el objeto puede reanudar su funcionamiento, ya sea en el mismo nodo o en un nodo distinto. La forma de conseguir un estado consistente en el objeto consiste en bloquear todas las invocaciones entrantes a métodos del objeto desde el exterior, permitir que las llamadas a métodos del objeto que estuvieran en curso terminen e, incluso, suspender la ejecución de algunas llamadas que estén en curso y no se prevea que vayan a finalizar en un corto plazo de tiempo.

El meta-objeto responsable del control de las llamadas a método entrantes (política de aceptación de mensajes) y de las que ya están en curso (suspender, reanudar y finalizar un método) es el **sincronizador** (ver capítulo 9).

Todo objeto susceptible de ser migrado dispone en su meta-espacio de un atributo de tipo lógico denominado `estadoMigración`, almacenado en su **meta-objeto datos**. El meta-objeto datos permite almacenar información privada acerca de su objeto base asociado para ser manipulada por los objetos que conforman su meta-espacio.

Cuando el objeto movedor desea llevar a un objeto a migrar a un estado consistente, invoca el método `obtenerEstadoMigración` del meta-objeto datos. La ejecución de dicho método suspende la ejecución de todos aquellos hilos susceptibles de serlo, deja terminar aquellos métodos que están terminando su ejecución y modifica el valor del atributo `estadoMigración`. Los métodos que se suspenden para poder migrar el objeto son anotados, de tal forma que puedan ser restaurados a su estado previo una vez que el objeto se encuentre en el nodo destino de la migración. La modificación del atributo `estadoMigración` permite bloquear todas las invocaciones entrantes al objeto base, dado que es consultado por el meta-objeto sincronizador antes de crear un nuevo hilo para ellas. Una vez que el objeto se encuentra en un estado consistente, el objeto movedor obtiene de nuevo el control.

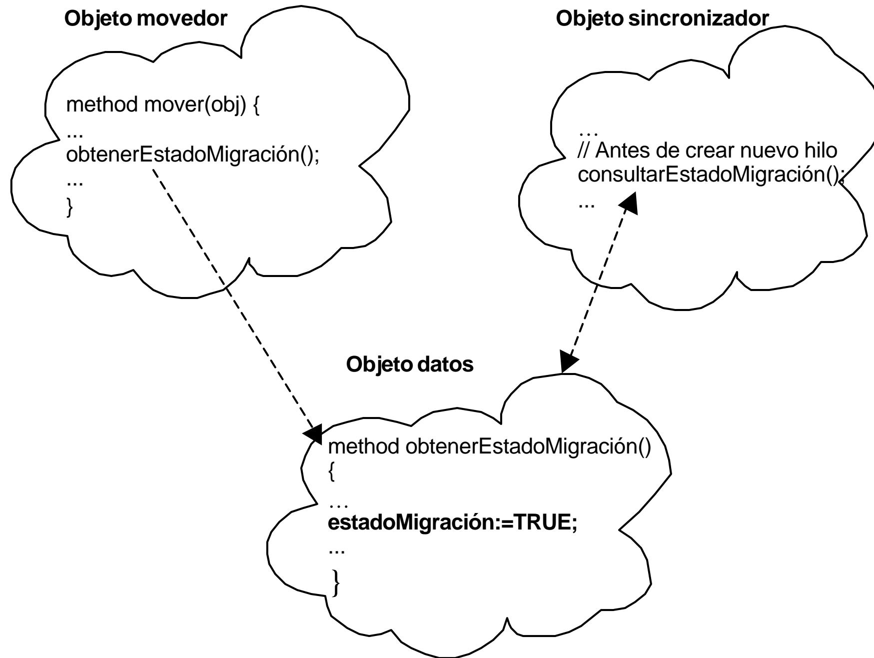


Figura 12.11. Relación entre los meta-objetos movedor, sincronizador y datos para la migración.

A continuación, el objeto movedor debe **obtener la copia serializada del objeto**, que consiste en una cadena de caracteres que representará el estado encapsulado del objeto. Dicha copia se obtiene invocando al objeto reflejado del área de instancias, que implementa dicha funcionalidad (`serializeInstance`).

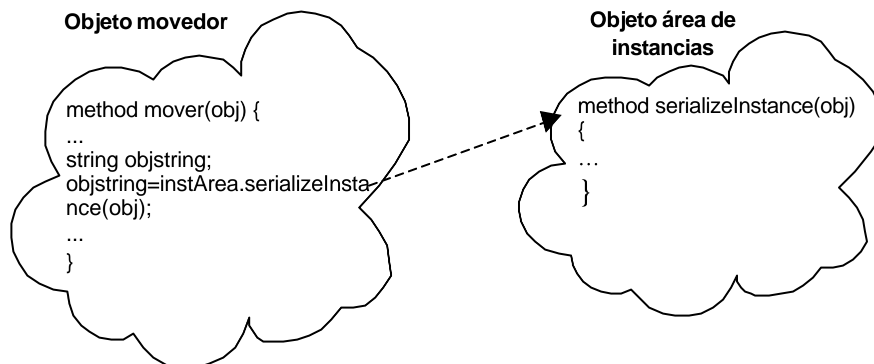


Figura 12.12. Obtención de la copia serializada de un objeto.

La copia serializada del objeto contiene la información siguiente:

- El identificador del objeto.
- Las referencias a todos los atributos del objeto, ya sean propios o heredados.
- La referencia a su clase.
- La clase sombra.
- Las referencias a sus meta-objetos compartidos.
- Los meta-objetos privados.

No se incluye como información a serializar ninguna instancia referenciada desde el objeto, dado que la decisión de mover dichas instancias tiene que ser tomada por la política de migración correspondiente (la que decide el qué). Aunque en un principio parece lógico pensar que los agregados del objeto deberían migrar con él, no es imprescindible. La política que decide qué mover será la responsable de ello.

La únicas excepciones a la regla anterior son la clase sombra y los meta-objetos asociados al objeto y que no lo están a ningún otro (privados).

Será necesario mover la clase sombra para posibilitar la activación inmediata del objeto en el nodo destino. En el caso de que el objeto no tuviera una clase sombra asociada (eso querría decir que toda la información de clases y superclases del objeto estarían disponibles localmente), se creará una *ad hoc*, que incluirá una copia de todos los métodos que estuvieran activos en el objeto al iniciarse la migración. Si el objeto no tenía clase sombra ni presentaba actividad interna en el momento de la migración, no se enviará clase sombra alguna.

Por otro lado, aquellos meta-objetos que recojan meta-información particular del objeto a migrar, migrarán con el propio objeto. De los objetos descritos, el único que cumple con dicha propiedad es el meta-objeto datos. El resto de los meta-objetos no se migran, dado que son compartidos por el conjunto de objetos del sistema operativo y se dispondrá de copias de los mismos en todos los nodos del sistema integral.

La figura siguiente muestra un ejemplo de creación de la copia serializada de un objeto que ha sido elegido para migrar, donde toda la información acerca del objeto está disponible localmente (en el propio nodo) y para el que se supone que existe al menos un hilo de ejecución activo en uno de sus métodos (`method1`).

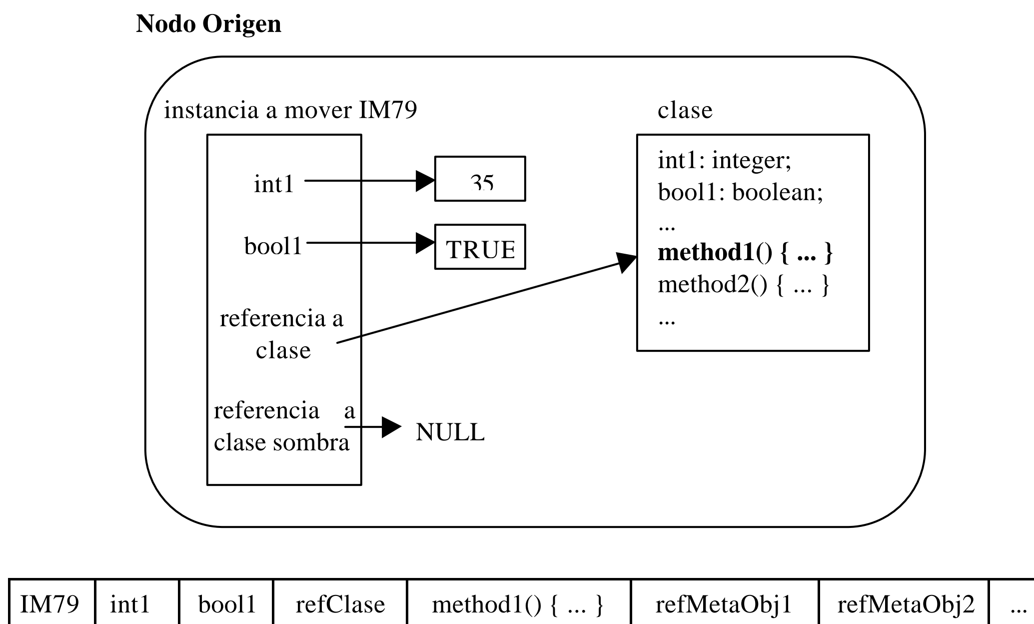


Figura 12.13. Copia serializada de un objeto con un hilo activo en un método.

12.4.4.2 Enviar la copia serializada del objeto

Una vez que el objeto movedor dispone de la copia serializada del objeto, procede a enviarla a la máquina virtual de destino. Para ello, solicita los servicios del objeto

comunicador, proporcionándole la identidad del nodo destino, el tipo de mensaje (migración de un objeto) y la cadena de caracteres que encapsula el objeto a mover.

En el caso de que se desease realizar una comunicación segura, se podrá realizar algún tipo de **cifrado de la información** con anterioridad a su entrega al objeto comunicador. La operación de descifrado se realizará de manera análoga en el extremo receptor del mensaje.

En la máquina virtual de destino se estará ejecutando otro objeto comunicador, que recibirá el mensaje, y se lo pasará a un objeto **trabajador** que se encargará de completar la operación de migración, como se muestra en los apartados siguientes.

En el caso de que la **operación de transmisión del objeto no se completara con éxito** (por ejemplo, por un problema en la red de comunicaciones), la operación de migración, a su vez, se consideraría fracasada, con lo que el objeto no se movería del nodo en el que reside. Para dejar las cosas tal y como estaban antes de la operación de migración, se invoca el método `restaurarEstadoNoMigración` de su meta-objeto datos. La invocación de dicho método activará de nuevo los hilos que fueron suspendidos para llevar al objeto a un estado consistente y modificará el valor del atributo `estadoMigración` para desbloquear las invocaciones entrantes al objeto

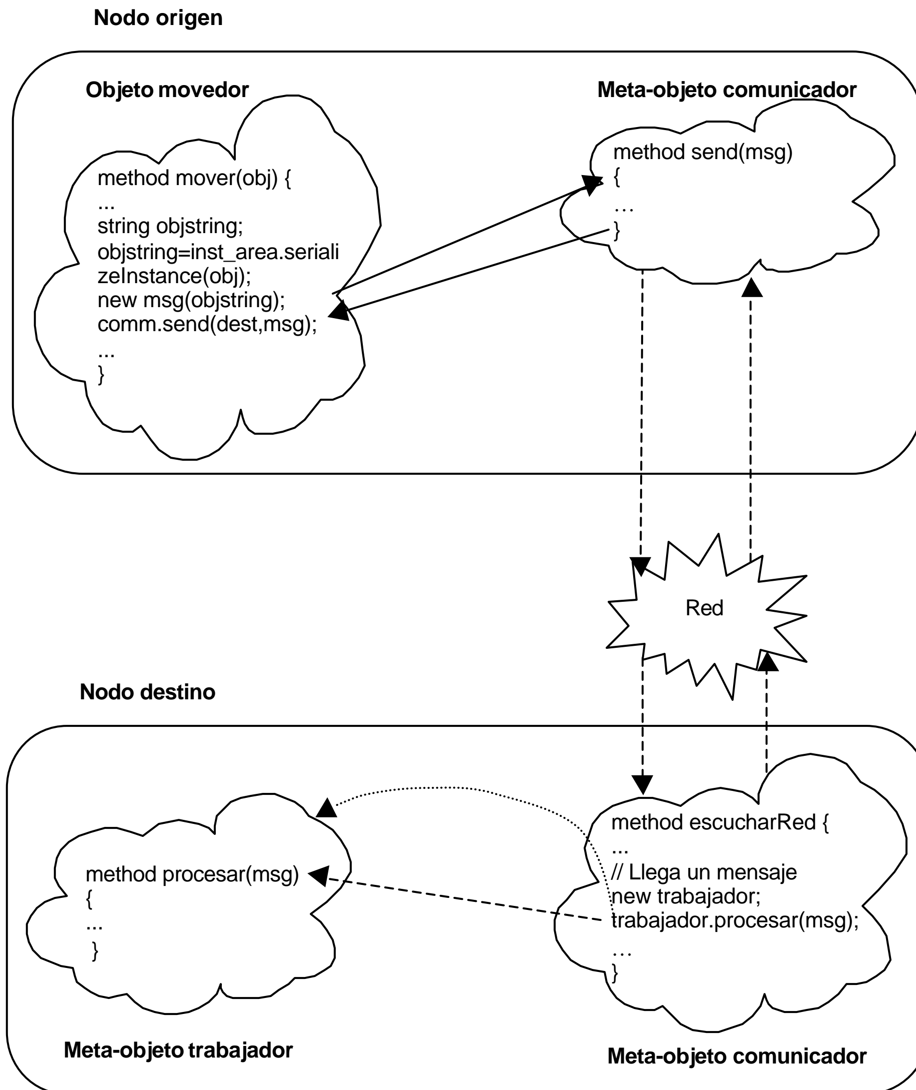


Figura 12.14. Envío de la copia serializada del objeto.

12.4.4.3 Creación de la instancia en el nodo de destino

Una vez que el nodo destino dispone de la copia serializada del objeto movido, es preciso convertir dicha copia en un objeto normal. El objeto trabajador se encargará de solicitar dicha acción al objeto reflejado del área de instancias, con una invocación al método `createObjectFromString`. El objeto reflejado del área de instancias creará la instancia solicitada, asignándole el identificador que ya tenía en el nodo original y dejándola en estado suspendido. Hay que hacer notar, en este punto, que **el objeto reflejado del área de instancias se salta el procedimiento de asignación de identificadores** a objetos ya enunciado, debido a la necesidad de que el objeto conserve su identidad allá donde quiera que se encuentre.

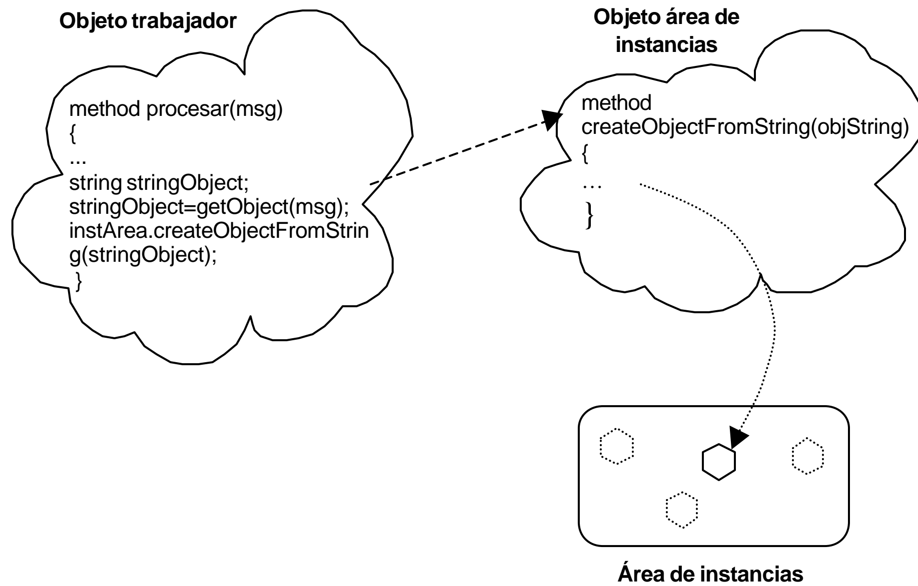


Figura 12.15. Creación del objeto a partir de su copia serializada.

La creación de la instancia a partir de la copia serializada debe recrear el estado original del objeto. En primer lugar, se creará la clase sombra del objeto, siempre y cuando sea necesario. Será necesario crear la clase sombra si se da alguna de las situaciones siguientes:

- La clase sombra no formaba parte de la copia serializada del objeto y la definición de la clase o alguna de las superclases del objeto no está disponible localmente. En este caso, el objeto no presentaba actividad interna en el nodo origen, por lo que la clase sombra se crea únicamente con los tipos de los atributos propios y heredados del objeto.
- La clase sombra formaba parte de la copia serializada del objeto e incluía algún atributo o método de alguna de las superclases del objeto que no está disponible localmente. La clase sombra se crea con los tipos de los atributos propios y heredados del objeto y la definición de los métodos que formaban parte de la clase sombra en la copia serializada del objeto.

En segundo lugar, se asignará espacio en el área de instancias para el objeto, espacio que albergará las referencias a los atributos del objeto (propios y heredados), y las referencias a la clase y la clase sombra.

12.4.4.4 Reactivación del objeto

Cuando la instancia se haya copiado con éxito en el nodo de destino, se realiza un conjunto de actividades que dan por finalizada la migración del objeto:

- **Eliminar la copia antigua del objeto**. La copia del objeto que reside en la máquina origen ha de ser eliminada, ya que en este momento existen dos copias exactas del mismo objeto. El objeto movedor solicitará al objeto reflejado del área de instancias la realización de esta acción.
- **Actualización de la información de localización** del objeto. La actualización de esta información no se puede realizar hasta este momento, dado que es ahora

cuando se tiene la seguridad de que se ha podido migrar el objeto con éxito. La forma en que la información de localización se va a actualizar depende, fundamentalmente, de la política de localización.

- **Reactivación del objeto**. Una vez que el objeto migrado ya se encuentra estable en el nodo remoto, llega el momento de activarlo, con el fin de que pueda reanudar sus tareas de cómputo. Para activar de nuevo el objeto, el objeto trabajador invocará el método `restaurarEstadoNoMigración` de su meta-objeto datos. La invocación de dicho método activará de nuevo los hilos que fueron suspendidos para llevar al objeto a un estado consistente (obtención del estado encapsulado del objeto) y modificará el valor del atributo `estadoMigración` para desbloquear las invocaciones entrantes al objeto.

El objeto se encuentra en este momento **en disposición de ser planificado** para ejecutar sus hilos activos tan pronto como sea elegido para ello. Toda la información de que disponía el objeto para su ejecución en el momento en que fue suspendido para migrar está disponible en su nueva ubicación. La única diferencia es que determinadas referencias que en su ubicación original apuntaban a instancias locales estarán apuntando ahora a instancias remotas. Será responsabilidad de las políticas de migración el minimizar los efectos (sobre todo, en el rendimiento del sistema) de esta nueva situación.

La figura siguiente muestra el estado final de la migración del objeto, tanto en el nodo origen como en el nodo destino de la migración, en cuanto a las estructuras de datos relativas al objeto migrado. Hay que hacer notar que las instancias que se corresponden con los atributos del objeto no han sido migrados, dado que no se desea suponer política de migración alguna, aunque, en general, se puede pensar que los atributos de un objeto se mueven con el propio objeto. Dado que el objeto ya no reside en el mismo nodo que su clase, ha sido necesario crear una clase sombra a partir de la copia serializada del objeto, en la que destaca la definición del único método que estaba en ejecución en el nodo origen.

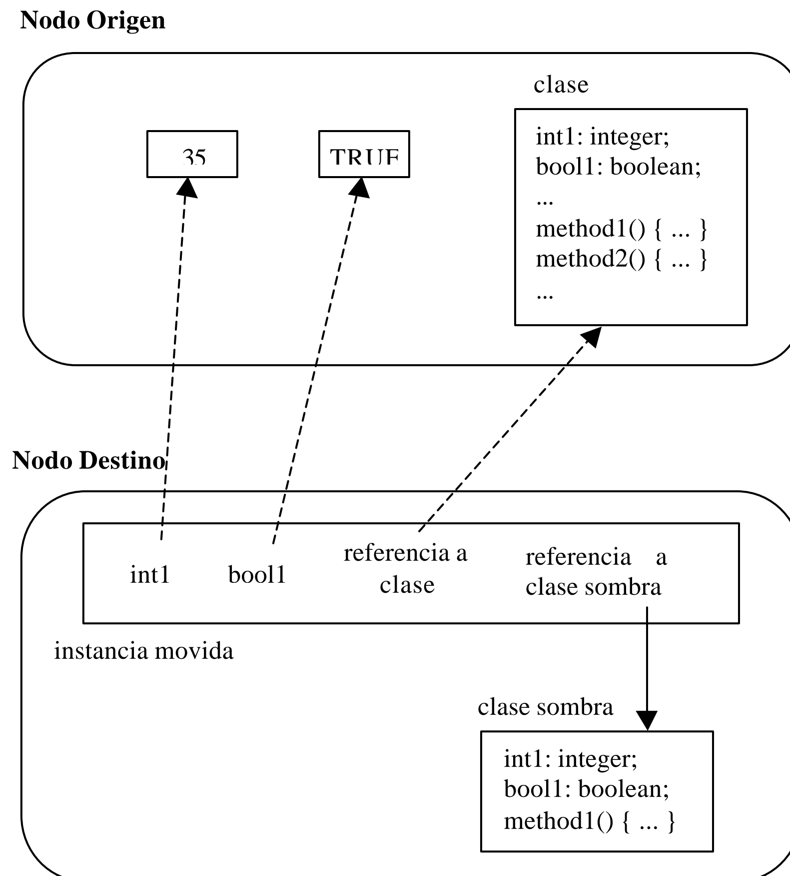


Figura 12.16. Creación de la instancia en el nodo destino de la migración.

12.4.4.5 Reenvío de mensajes

Cuando un objeto se encuentra en proceso de migración, puede seguir recibiendo invocaciones de aquellos objetos que aún no conocen su nueva situación. Es necesario que dichas invocaciones no se rechacen o pierdan, sino que se reenvíen al objeto tan pronto como esté preparado en su destino, con el fin de proporcionar una total transparencia en el proceso de migración.

El meta-objeto receptor del objeto migrado es el primer conocedor de las invocaciones entrantes al objeto. Antes de proceder a pasarle el control al meta-objeto sincronizador para que cree el hilo para la invocación, consultará el estado del atributo `estadoMigración`, que le permitirá saber que el objeto se encuentra en una situación transitoria. El meta-objeto receptor retrasará la entrega de la invocación al meta-objeto sincronizador hasta el momento en que el objeto esté activo en el nodo destino. En ese momento, y a través de la red, le hará llegar toda la información acerca de las invocaciones que llegaron al objeto mientras estaba migrando.

12.4.5 Cuándo, qué y adónde migrar objetos

Una vez visto el mecanismo básico de migración (cómo mover objetos) quedan por **establecer las políticas de migración**. Se trata de responder a la pregunta: cuándo mover qué objetos y adónde. De nuevo, y como ya ha ocurrido en otros mecanismos del sistema de distribución (por ejemplo, en la localización de objetos), las políticas de migración vienen descritas por un conjunto de objetos. Con el fin de conseguir la

deseada independencia entre mecanismos, los objetos que implementan cada una de las políticas deben ser a su vez independientes.

Sean cuales sean las políticas de migración de objetos utilizadas, tienen que asegurar que un objeto pueda estabilizarse durante determinados periodos de tiempo en una máquina abstracta, con el fin de llevar a cabo su propia computación (la ejecución de sus métodos ante invocaciones externas). También es importante que la política de migración se ponga de acuerdo con la política de replicación (si es que existe), con el fin de que no coexistan diferentes réplicas del mismo objeto en la misma máquina abstracta.

El diseñador de las políticas de migración deberá tener en cuenta todos estos aspectos, con el fin de que la migración no degrade el rendimiento del sistema, en contra de su objetivo inicial.

12.4.5.1 Cuándo

La decisión de cuándo mover un objeto viene determinada por una de las razones siguientes: una solicitud explícita de migración o bien, un proceso previo de recolección de estadísticas sobre el comportamiento de los objetos.

Solicitud explícita de migración

La solicitud explícita de migración es utilizada por aquellos objetos que son conscientes de la arquitectura distribuida del sistema integral y desean sacar partido de dicho conocimiento para su computación. La solicitud es realizada por dichos objetos al **objeto movedor** local.

También se puede considerar como una solicitud explícita relacionada con la migración la petición de un objeto para ser fijado en un nodo determinado. De la misma manera, los objetos movedor ofrecen una operación para liberar un objeto previamente fijado.

En el capítulo 10 se muestran varios ejemplos de solicitudes de migración explícitas en el paso de parámetros y la devolución de resultados en la invocación remota de métodos.

Estudio del estado del sistema

Adicionalmente y de manera periódica o ante la aparición de determinado evento, se realiza un proceso de recolección de estadísticas sobre el comportamiento de los objetos. Las estadísticas recogidas son estudiadas y en base al resultado de dicho estudio se toma la decisión de mover objetos. Las estadísticas son recogidas básicamente para determinar:

- La carga de la máquina abstracta, medida en número de objetos, tamaño de los objetos, número de hilos, etc.
- La carga de comunicaciones con otras máquinas abstractas, medida, por ejemplo, en número de mensaje por unidad de tiempo.

Los objetos del sistema operativo serán los encargados de recoger toda la información necesaria para la toma de decisiones. La carga de la máquina abstracta se puede determinar consultando a los objetos reflejados del área de instancias y el área de hilos. La carga de las comunicaciones puede ser computada por parte de los objetos comunicador.

12.4.5.2 Qué

Se ha definido el objeto como la unidad elemental de migración. En ocasiones, puede no ser rentable mover objetos individuales. Por ejemplo, los objetos primitivos o elementales de los lenguajes de programación, como los enteros o los reales, son demasiado pequeños como para ser movidos de manera individual.

Es necesario, por tanto habilitar una **política de agrupación de objetos** para decidir qué objetos se tienen que mover conjuntamente [BHD+98]. Dichas políticas trabajan sobre conjuntos de objetos que guardan entre ellos algún tipo de relación.

Relaciones básicas

Al nivel de la máquina abstracta, los objetos pueden presentar ciertas relaciones entre ellos:

- Relación de agregación o “es parte de”. Es lógico pensar que los agregados (o partes de) un objeto se mueven con el propio objeto. La implementación de la política de agrupación en este caso es muy sencilla, dado que lo único que hay que hacer es mover recursivamente todos los objetos agregados de uno que ya se ha decidido mover.
- Relación de asociación genérica. Se utiliza para que los objetos puedan mantener relaciones que no entran en el apartado anterior. Aunque en principio, la implementación de la política de agrupación utilizando estas relaciones sería sencilla, habría que preguntarse si se mueven siempre todos los objetos asociados.

Relaciones de alto nivel

Además de las relaciones heredadas del propio modelo de objetos, sería posible establecer relaciones a más alto nivel, es decir, al nivel del sistema operativo. El sistema operativo puede permitir a los objetos especificar de qué manera desean agruparse a efectos de su migración (nótese que esto tendría únicamente sentido para aquellos objetos conscientes de la característica distribuida del sistema integral), a la manera de los *group* de Emerald [Jul88] o de los *cluster* de FlexiNet [HBD+98].

Otras consideraciones

Las relaciones existentes entre los objetos no son lo único a tener en cuenta para decidir qué objetos mover. Es necesario además tener en cuenta:

- El tamaño total de los objetos a mover. La cantidad de información a enviar por las red de comunicaciones puede ser demasiado elevada para que merezca la pena realizar la migración. Hay que tener en cuenta que el coste de las comunicaciones es relativamente elevado.
- El tiempo estimado para acabar el procesamiento de los objetos considerados. Si un objeto a migrar va a finalizar en un breve plazo de tiempo su computación, puede ser más interesante no migrarlo, puesto que puede ser mas corto dicho plazo que el tiempo que se necesite para moverlo.
- El número de veces que ya ha sido movido cada objeto. Los objetos que se mueven muchas veces no tienen la oportunidad de ejecutarse como los demás. Por tanto, si un objeto no se ha ejecutado prácticamente desde que se movió la última vez, sería conveniente descartarlo para migrar una vez más.

- Si la migración se ha solicitado explícitamente, es necesario evaluar el estado del nodo destino de la migración. El sistema operativo no puede atender siempre las peticiones de migración explícitas, ya que en algunos casos podrían ir en contra del equilibrio de carga de los nodos del sistema integral. En estos casos, el objeto que solicitó la migración será informado de la imposibilidad de realizarla.

Por tanto, la decisión de qué conjunto de objetos mover en un momento determinado debe resultar de una **evaluación ponderada** de todas las cuestiones anteriores, pudiéndose llegar incluso a la conclusión de que no es rentable realizar migración de objetos alguna. La ponderación de todas las consideraciones descritas tendrá que ser realizada por el diseñador del sistema operativo.

12.4.5.3 Adónde

Una vez se han escogido los objetos a migrar, es necesario decidir cuál va a ser su destino. La elección del destino depende en gran medida de la razón por la que se desea migrar el objeto:

- El objeto se va a migrar porque ha sido solicitado de manera explícita. El destino de la migración será el nodo que se haya especificado en la petición. En caso de que, por alguna causa (por ejemplo, una elevada carga de computación), el nodo destino no pueda aceptar un nuevo objeto, se denegará la migración.
- El objeto se va a migrar para equilibrar la carga de los diferentes nodos del sistema integral. Será preciso evaluar la carga actual de cada uno de los nodos, el "tamaño" de cada uno de los objetos a mover y el coste de moverlos a cada uno de los posibles destinos para elegir el destino final de cada uno de ellos.
- El objeto se va a migrar porque presenta un alto grado de interacción con algún objeto remoto. De nuevo el destino de la migración es fácil de deducir. Se moverá el objeto al nodo en el que reside el objeto remoto, de manera que se minimice el número de mensajes que circulan por la red para atender invocaciones remotas. En el caso de que el nodo remoto deniegue la migración por estar muy cargado, se podría realizar una migración a un nodo cercano a él. El registro del grado de interacción de objetos locales con objetos remotos se puede realizar en la consulta al objeto localizador, que es utilizado siempre que se produce una invocación a un objeto.

Excepto en el caso de la migración explícita, y tal como ocurría en el apartado dedicado a cuándo migrar, **la elección del nodo destino de la migración estará sujeta a la disponibilidad de datos estadísticos que describan el estado actual del sistema integral**. De cualquier otra manera, la migración se haría a ciegas y con muchas posibilidades de degradar el rendimiento del sistema.

12.5 Gestión de fallos de nodos

Un sistema centralizado, o bien se está ejecutando, o bien ha fallado y no lo está haciendo. En contraste, un sistema distribuido formado por un conjunto de nodos autónomos puede fallar parcialmente, existiendo cierto conjunto de nodos en perfecto funcionamiento y otro conjunto de ellos que está parado.

El sistema integral distribuido es de este último tipo de sistema. Es necesario detectar aquellos nodos que han fallado, con el fin de restringir las actividades del sistema a aquellos que funcionan adecuadamente. Por ejemplo, un objeto que se

encargue del equilibrado de carga del sistema integral deberá tener en cuenta únicamente aquellos nodos que funcionan, ignorando a los que han fallado.

Se dice que un **nodo está disponible** cuando está ejecutándose; en otro caso, está **no-disponible**. De la misma manera, un **objeto está no-disponible** si no se puede encontrar en ninguno de los nodos disponibles.

Cuando se invoca un objeto no-disponible, la invocación falla. La no-disponibilidad del nodo en el que reside el objeto es detectada por el meta-objeto comunicador, que falla al intentar ponerse en contacto con el meta-objeto comunicador remoto para enviarle el mensaje de la invocación. El meta-objeto comunicador local elevará una excepción, que será recogida por el meta-objeto emisor y propagada al objeto base cliente, con el fin de que maneje la situación errónea que se ha producido.

En el caso de la migración, la no-disponibilidad de determinados nodos imposibilita que sean elegidos como destino del movimiento de los objetos. Las diferentes políticas de migración tendrán que adaptarse dinámicamente a la disponibilidad de nodos en el sistema integral. Por otro lado, la no-disponibilidad de un nodo puede producirse durante el proceso de migración. Si el problema se produce en el nodo destino de la migración, la solución al problema pasa por reactivar el objeto en el nodo origen, invocando al método `restaurarEstadoNoMigración` de su meta-objeto datos. Si el problema se produce en el nodo origen de la migración y el estado encapsulado del objeto ya ha sido transferido totalmente al nodo destino, la migración se completa sin la eliminación de la copia del objeto ni la actualización de la información de localización en el nodo origen, que ya no son necesarias.

12.6 Resumen

La invocación remota y la migración de objetos son las dos facilidades básicas de todo sistema de distribución de objetos. En el sistema integral, dichas facilidades son proporcionadas por un conjunto de meta-objetos que sobrescriben determinados comportamientos establecidos por defecto en las máquinas abstractas.

La invocación remota de métodos reposa principalmente en los meta-objetos emisor y receptor de los objetos cliente y servidor, respectivamente, de la invocación. El meta-objeto emisor solicita al objeto comunicador local que envíe el mensaje con los parámetros de la invocación al nodo remoto. Allí, otro objeto comunicador recibe el mensaje, creando un objeto trabajador que completa la invocación al objeto servidor, ya de manera local. Posteriormente, y siguiendo una secuencia inversa, se devuelve el resultado de la invocación. Los objetos cliente y servidor son totalmente ajenos a todo el conjunto intermedio de operaciones.

La migración de objetos es una facilidad del sistema operativo que intenta maximizar el rendimiento global del sistema integral distribuido. De todas formas, y en determinados casos, se puede permitir a los objetos de usuario que soliciten la migración de objetos. La operación de migración se realiza, en general, para un conjunto de objetos seleccionados al efecto de manera previa. Cada uno de los objetos migra siguiendo un patrón concreto: dejar el objeto en un estado consistente; obtener una copia de su estado encapsulado; enviar dicha copia al nodo destino de la migración; creación del objeto en el nodo destino y reactivación del objeto y eliminación de la copia antigua en el nodo origen.

CAPÍTULO 13 ASPECTOS ADICIONALES RELACIONADOS CON LA DISTRIBUCIÓN

13.1 Introducción

Aunque los aspectos de los que se habla en este capítulo son importantes, no se han considerado como básicos para la distribución de objetos. Su inclusión en el sistema operativo distribuido orientado a objetos permite ofrecer un mejor servicio a los usuarios. Será responsabilidad del implantador del sistema incluir los servicios de nombrado, transacciones y replicación en función del entorno en el que vaya a ejecutarse. De nuevo, la reflectividad se presenta como la herramienta que posibilita la extensión del sistema operativo de una manera sencilla para la introducción de nuevos servicios.

13.2 Servicio de nombrado

El **servicio de nombrado** o servicio de directorio o servicio de denominación, es proporcionado al nivel del sistema operativo para proporcionar atributos de entidades de un sistema distribuido a partir de un nombre. Las entidades pueden ser de cualquier tipo y pueden estar gestionadas por diferentes servicios. La asociación de un nombre con una entidad se denomina **vínculo** (*binding*).

Un servicio de nombrado gestiona una base de datos de vínculos entre un conjunto de nombres textuales (simbólicos o legibles) y atributos de entidades. Los atributos que se almacenan para las entidades deben ser escogidos para ser útiles, no sólo para los usuarios, sino para otros servicios del sistema distribuido.

Nombre	Atributos		
	UID	Nombre largo	Shell
root	0	Administrador del sistema	/bin/csh
falvarez	432	Fernando Álvarez García	/bin/bash
...	...	...	...
aperez	701	Antonio Pérez del Río	/bin/bash

Figura 13.1. Relación entre nombres y atributos.

La operación principal de todo servicio de nombrado es la de **resolver** un nombre, es decir, buscar en la base de datos los atributos para un nombre dado. También son necesarias operaciones para la creación de nuevos vínculos para nombres nuevos, la eliminación de vínculos y el listado de nombres vinculados.

Aunque se hable de que el conjunto de vínculos se organiza en una base de datos, es importante indicar que los nombres no son siempre claves simples, sino que más bien están formados por un conjunto de componentes (por ejemplo,

usuario@pinon.ccu.uniovi.es), que deben ser buscados en partes separadas de la base de datos, denominadas **contextos**.

Un aspecto muy llamativo de los sistemas distribuidos es que los servicios que ofrece tienen, en general, un nombre. En los sistemas centralizados no nos encontramos, en general, un caso equivalente, dado que el servicio esta implícito en la llamada al sistema que se invoca.

En los sistemas distribuidos modernos el servicio de nombrado está implementado como una **base de datos distribuida, replicada y orientada a objetos**. Es distribuida para permitir a diferentes dominios de administración controlar su entorno. Se replica con el fin de proporcionar una buena disponibilidad y rendimiento, allá donde sean necesarios. Finalmente, es una base de datos orientada a objetos en el sentido de que todo lo que se registra es una instancia de una clase. Se puede, por tanto, utilizar la herencia para derivar nuevos tipos de objetos.

Los servicios de nombrado modernos proporcionan una **interfaz** que permite a sus clientes (programas o usuarios) localizar entidades en el sistema distribuido a partir de su nombre o atributos. El propio servicio de directorio debe residir en una **dirección bien conocida** por todos sus potenciales usuarios.

13.2.1 Servicio de nombrado para el sistema integral

El servicio de nombrado del sistema integral no tiene por qué tener ninguna funcionalidad especial. Bastaría con un servicio de nombrado mínimo, centralizado, con un espacio de nombres plano, y cuya interfaz proporcionase únicamente las operaciones siguientes: crear y eliminar un vínculo para un objeto y resolver un nombre. De todas formas, es posible construir un servicio de nombrado distribuido y replicado, que recoja diferentes atributos de cada uno de los objetos registrados.

13.2.1.1 Servicio de nombrado básico

El término centralizado quiere decir que el servicio de nombrado estaría ubicado en un único nodo del sistema integral, de manera que en dicho nodo se almacenaría la base de datos de vínculos y todas las peticiones al servicio de nombrado tendrían que dirigirse a dicho nodo. Básicamente, el servicio de nombrado estaría formado por un objeto servidor de nombres, al que se dirigirían todas las peticiones, y un conjunto adicional de objetos que mantendrían la base de datos de vínculos.

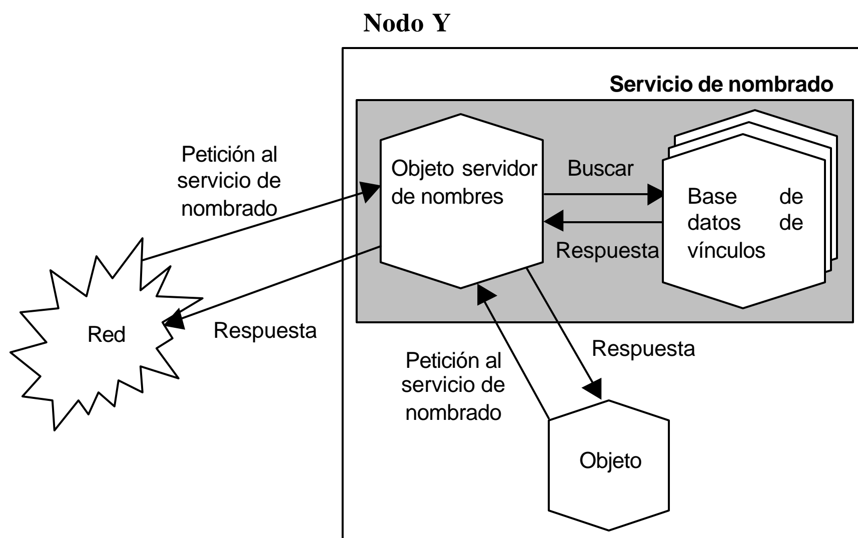


Figura 13.2. Servicio de nombrado elemental.

En segundo lugar, el espacio de nombres sería plano, es decir, los nombres tendrían una única componente (por ejemplo, “impresora”, “usuario1”, etc.).

La operación de creación de un nuevo vínculo necesitaría, en esta versión básica, el nombre simbólico del objeto (un *string*) y una referencia al mismo, que será el valor devuelto por la operación de resolución de un nombre.

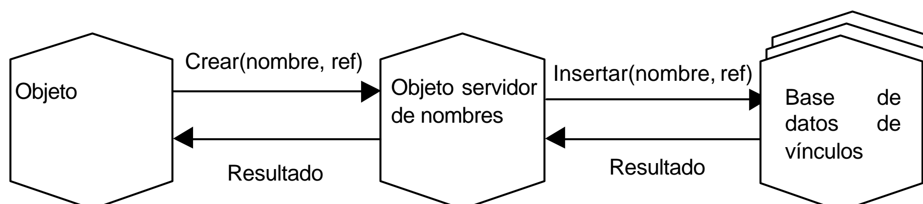


Figura 13.3. Creación de un vínculo en el servicio de nombrado.

El servidor de nombres sólo va a permitir crear un vínculo al propio objeto apuntado por la referencia pasada como parámetro. Para ello, el objeto servidor de nombres sólo tendrá que comprobar que la identidad del objeto contenida en la referencia se corresponde con la identidad del objeto que le invoca (en el capítulo 12 se muestra cómo forma siempre parte del mensaje de la invocación una referencia al objeto invocador). Por tanto, el objeto que solicita la creación de un nuevo vínculo es el único responsable de los permisos que va a proporcionar a sus potenciales clientes a través de la capacidad.

La operación de eliminación de un vínculo necesita, únicamente, el nombre del objeto a eliminar. De nuevo, el objeto servidor de nombres comprobará que el objeto que demanda la operación se corresponde con el que está almacenado en la base de datos para el nombre especificado.

Finalmente, la operación de resolver nombre podrá ser demandada por cualquier objeto para cualquier nombre. El objeto servidor de nombres se encargará, únicamente, de recuperar la referencia asociada al nombre proporcionado en la base de datos de vínculos y retornar una copia de la misma al objeto solicitante.

13.2.1.2 Servicio de nombrado distribuido y replicado

Para construir un servicio de nombrado distribuido y replicado en el sistema integral es necesario, en primer lugar, repetir el esquema de objeto servidor y conjunto de objetos que forman la base de datos de vínculos en varios nodos del sistema integral.

Cada objeto servidor local dispondrá de referencias a los servidores remotos. El hecho de que una referencia sirva para apuntar a un objeto con independencia de su localización, permite que los servidores de nombres puedan incluso llegar a ignorar la ubicación exacta de cada uno de los otros (sólo les va a interesar que existen y qué pueden hacer si los necesitan). La interfaz que proporcionan los servidores tendrá que ser aumentada para posibilitar la interacción entre todos ellos, de tal forma que puedan intercambiar información sobre vínculos siguiendo las políticas de distribución y replicación de nombres que se hayan establecido. Los nuevos métodos de la interfaz sólo podrán ser invocados por objetos servidores de nombres, de tal manera que un objeto de usuario tendrá que utilizar los métodos habituales.

Por su parte, será necesario proporcionar a todo objeto una referencia a alguno de los servidores de nombres disponibles. En general, se le proporcionará una referencia a su servidor de nombres local.

Una petición al servicio de nombrado será satisfecha, en un primer momento, por el servidor de nombres local. En el caso de que no pueda hacerlo (por ejemplo, no encuentra el nombre buscado en su base de datos local), solicitará la ayuda de otros servidores de nombres, de manera transparente para el objeto cliente (ver figura siguiente).

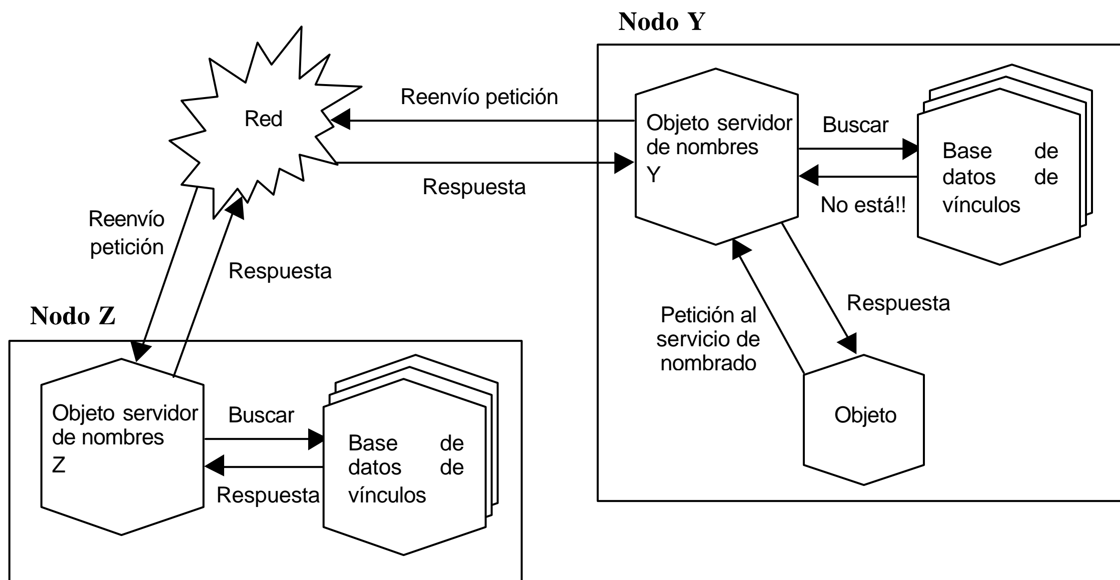


Figura 13.4. Servicio de nombrado distribuido y replicado.

13.3 Transacciones

Desde el punto de vista de los negocios, una transacción es una acción que cambia el estado de una empresa (por ejemplo, una transacción bancaria, en la que un cliente deposita dinero en una cuenta). Desde un punto de vista técnico, una **transacción** es una colección de acciones empapadas de las denominadas **propiedades ACID**, término acuñado en [HR83], que viene de las palabras inglesas *Atomicity*, *Consistency*, *Isolation*

y *Durability* (atomicidad, consistencia, aislamiento y durabilidad). A continuación se explica lo que significa cada una de ellas:

- **Atomicidad** significa que una transacción es la unidad indivisible de trabajo: todas sus acciones o tienen éxito o fallan, es decir, es una proposición todo-o-nada. La atomicidad se define desde la perspectiva del consumidor de la transacción.
- **Consistencia** significa que después de que se haya ejecutado la transacción, debe haber dejado el sistema en un estado consistente o haber abortado. Si la transacción no puede conseguir un estado estable final, debe dejar el sistema en su estado inicial.
- **Aislamiento** significa que el comportamiento de la transacción no puede verse afectado por otras transacciones que se ejecutan concurrentemente. La transacción debe serializar todos los accesos a los recursos compartidos y garantizar que varios programas concurrentes no corrompan entre ellos sus operaciones. Por ejemplo, un programa multiusuario que se ejecute bajo la protección de una transacción debe comportarse exactamente igual que en un entorno monousuario. Los cambios que realiza una transacción sobre los recursos compartidos no podrán ser visibles fuera de la transacción hasta que esta haya sido entregada (*commit*).
- **Durabilidad** significa que los efectos de la transacción son permanentes después de que haya sido entregada. Sus cambios deben sobrevivir a fallos del sistema. Podemos utilizar el término **persistente** como sinónimo de duradero.

Un servicio de transacciones debe registrar los efectos de todas las transacciones entregadas y ninguno de los efectos de las transacciones que han abortado. El servicio de transacciones debe, por tanto, estar al corriente de aquellas transacciones que abortan, tomando las precauciones necesarias para que no afecten a otras transacciones que se estén ejecutando concurrentemente.

En un mundo ideal, todas las aplicaciones estarían construidas en base a transacciones. Todos los programas tendrían que seguir una férrea disciplina transaccional, dado que con que un solo programa fallase, podría corromperse todo el sistema. Una transacción que trabaja inconscientemente sobre datos corrompidos, producidos por un programa no transaccional, está construida sobre unos fundamentos a su vez corrompidos.

13.3.1 Transacciones software

El coste de un servicio de transacciones que funcione constantemente para todas las aplicaciones de un sistema distribuido es muy elevado, dado que se tienen que garantizar las propiedades ACID para todas ellas. En los sistemas reales se tiende a marcar de alguna manera cuáles son las fronteras de toda transacción, es decir, dónde empieza y termina el conjunto de operaciones que definen una transacción en particular.

Adicionalmente, todo servicio de transacciones tiene que tener soluciones para las siguientes cuestiones:

- Cuándo se hacen accesibles los efectos de una transacción.
- Qué unidades de recuperación se utilizarán en el caso de fallos.

Existen diferentes modelos de transacciones que de una u otra manera solucionan los problemas planteados: transacciones planas, transacciones encadenadas y transacciones anidadas. En general, todas ellas tienen una variante distribuida. Con el fin de no hacer de esta sección un estudio en profundidad de las transacciones, se describirán las transacciones planas, que servirán como base para un estudio posterior de su incorporación al sistema integral.

Una **transacción plana** es aquella en la que las fronteras que marcan el inicio y fin de las operaciones están al mismo nivel.

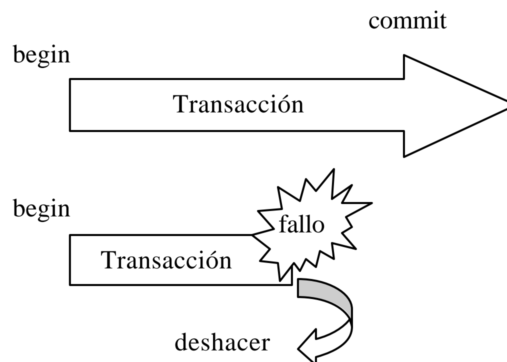


Figura 13.5. Transacción plana.

La transacción empieza con un *begin_transaction* y finaliza con un *commit_transaction* o con un *abort_transaction*. Todas las acciones entre el inicio y el final se realizarán de manera indivisible.

Los programas que se construyen en base a transacciones planas están divididos en un conjunto relativamente amplio de transacciones con un pequeño número de operaciones, que se ejecutan una tras otra.

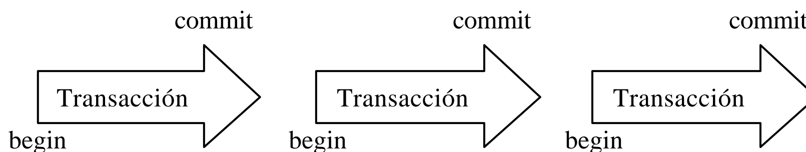


Figura 13.6. Construcción de un programa como conjunto de transacciones.

Una **transacción plana distribuida** es aquella que permite que alguna o varias de las operaciones que la componen utilice recursos de diferentes nodos del sistema distribuido. El servicio de transacciones se hace notablemente más complejo, siempre con el objetivo de hacer este incremento de la complejidad transparente al programador.

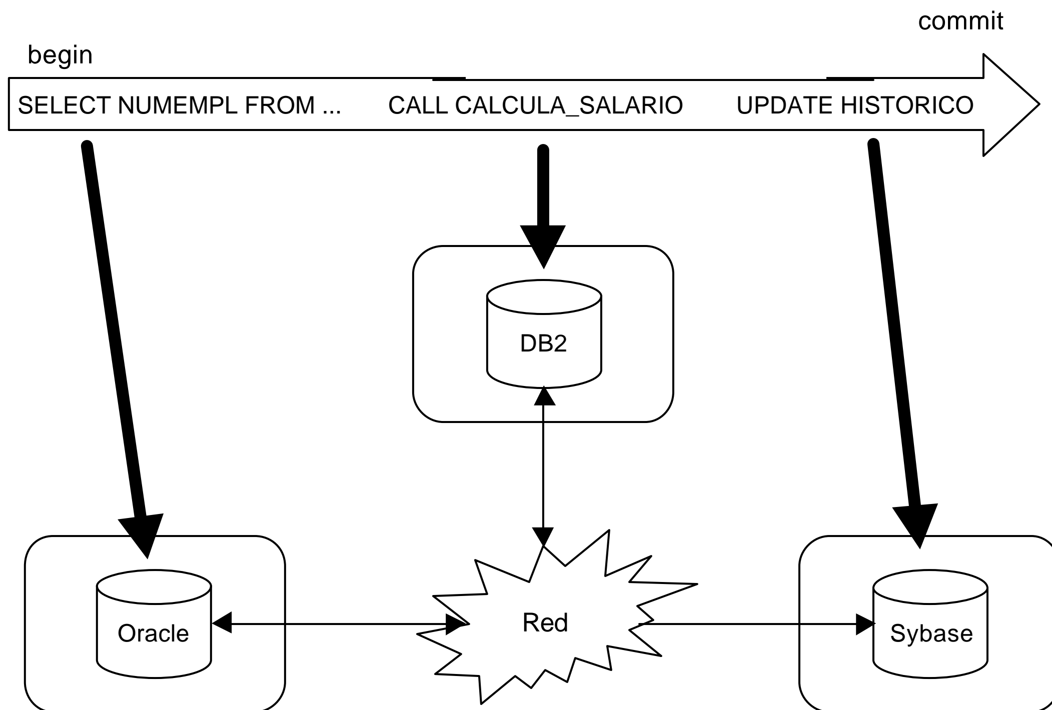


Figura 13.7. Transacción plana distribuida.

El servicio de transacciones deberá expandirse por todos los nodos del sistema distribuido para hacer posibles las transacciones distribuidas y en uno de ellos deberá coordinarse el conjunto de actividades para asegurar las propiedades ACID de la transacción. Todo esto se consigue con la utilización de un **protocolo de entrega de dos fases** (*two-phase commit protocol*) [CDK94], que coordina la entrega o la cancelación de una transacción que involucra varios nodos.

13.3.2 Introducción de transacciones en el sistema integral

Un servicio de transacciones para el sistema integral orientado a objetos debe **mantener las propiedades ACID para conjuntos de invocaciones** a objetos (las operaciones posibles en el sistema integral). El código de los objetos cliente y objetos servidor no debe verse alterado por la introducción de las transacciones más allá del etiquetado de inicio y fin de transacción (los objetos cliente serán los que demandarán la realización de transacciones, que involucrarán a uno o más objetos servidor). Por lo tanto, los objetos cliente contendrán únicamente la lógica del problema a resolver. La reflectividad del sistema permitirá introducir la funcionalidad necesaria para implantar el sistema de transacciones de manera transparente para los objetos de usuario.

El servicio de transacciones será el encargado de invocar a los objetos servidores en el momento adecuado y en la secuencia adecuada en base a las solicitudes realizadas por los objetos cliente. Para ello, interceptará todas las invocaciones a objetos que se realicen entre los límites de inicio y fin de transacción. Dicha interceptación es posible en dos puntos: en el meta-objeto emisor, del objeto cliente y en el meta-objeto receptor, del objeto servidor.

En la figura siguiente se muestra como sería posible implantar un sistema de transacciones en el sistema integral. El servicio de transacciones es invocado de manera transparente desde los meta-objetos emisor ② que obtienen el control en cada invocación a método ①. Está compuesto por un conjunto de objetos encargados de

mantener las propiedades ACID para todas las invocaciones que tienen lugar entre el inicio y el fin de la transacción y que involucran a un conjunto de objetos servidores. Con ese fin, pueden apoyarse en el servicio de persistencia del sistema integral ③. Finalmente, con el fin de hacer los cambios definitivos en los objetos servidores, el servicio de transacciones dialoga con los meta-objetos receptores ④ responsables de sus invocaciones entrantes ⑤.

El servicio de transacciones aquí mostrado proporciona un grado más de indirección en la invocación a métodos. El objeto de usuario es consciente únicamente de una invocación como la señalada en la figura con una **A**. Pero lo que realmente ocurre es que el meta-objeto emisor intercepta la invocación en primera instancia, (podría ser, incluso, una invocación remota) y el servicio de transacciones lo hace en segunda instancia, para garantizar la correcta realización de las operaciones sujetas a la transacción.

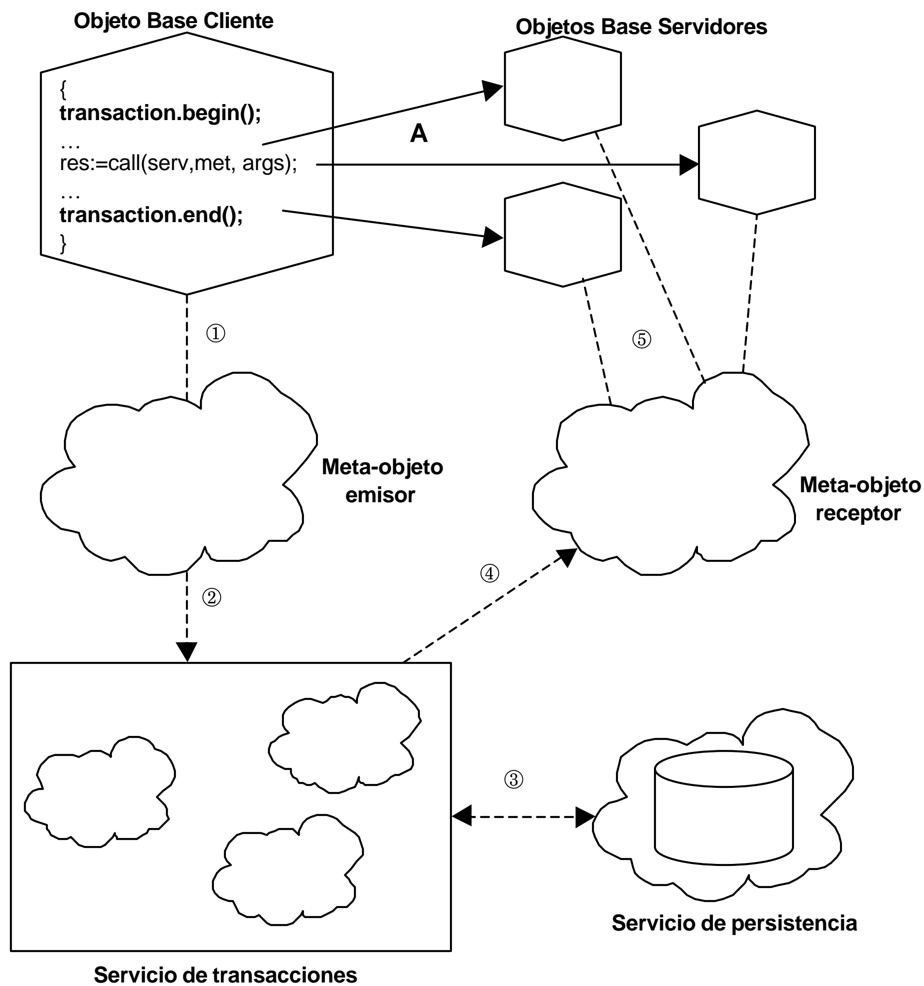


Figura 13.8. Diseño de un sistema de transacciones para el sistema integral.

13.4 Replicación

Un **servicio de replicación de objetos** permite la existencia de copias de un objeto en múltiples nodos del sistema distribuido. La primera motivación para la replicación es el incremento de la accesibilidad a un objeto, replicándolo en todos aquellos nodos en los que existen clientes suyos. Una segunda motivación es la tolerancia a fallos. Si se

replica un objeto en varios nodos, sus servicios serán más robustos ante posibles fallos en algún nodo. En ambos casos, una cuestión fundamental a resolver es el mantenimiento de la consistencia entre réplicas, es decir, asegurar que todas las réplicas mantienen los mismos valores, manteniendo unos tiempos de respuesta razonables. La motivación final para la replicación, como no podía ser de otra manera, es la mejora del rendimiento del sistema.

El requerimiento principal de cualquier esquema de replicación es el de la transparencia. Los clientes no tienen que estar al tanto de la existencia de múltiples copias, de tal forma que utilizarán un único nombre para referirse al objeto replicado, independientemente de la réplica concreta que vaya a servir sus peticiones. Otro requerimiento muy importante es el de la consistencia, que ya se ha comentado.

En la figura siguiente se muestra un modelo general de un sistema de replicación. Los objetos cliente realizan una serie de peticiones, que pueden ser de lectura o pueden actualizar información de las réplicas. Todas las peticiones de los clientes son gestionadas inicialmente por objetos *front-end*, encargados de la comunicación con las réplicas en nombre de los clientes. Dependiendo del esquema de replicación utilizado, los objetos *front-end* se comunican con las réplicas de diferentes formas.

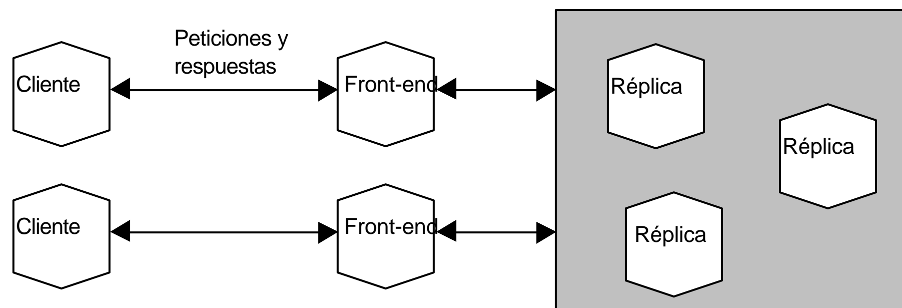


Figura 13.9. Arquitectura básica de un sistema de gestión de réplicas.

Existen diferentes esquemas de replicación de objetos: replicación únicamente de objetos inmutables, replicación de copia primaria y replicación de objetos iguales.

La **replicación de objetos inmutables** es la más sencilla. Los objetos inmutables son aquellos que, una vez creados, sólo pueden ser consultados por objetos cliente. Desaparece, por tanto, el problema del mantenimiento de la consistencia. Su aplicación, sin embargo, no es muy amplia en un sistema distribuido, dado que el número de objetos compartidos inmutables tiende a ser muy escaso.

En un esquema de **replicación de copia primaria** una réplica del objeto es etiquetada como la copia primaria, mientras que el resto de copias se ordenan y mantienen en el resto de los nodos como copias secundarias. Las peticiones que no modifican el estado del objeto pueden ser satisfechas por cualquier réplica. Las peticiones que modifican el estado del objeto son servidas únicamente por la copia primaria, que propaga las modificaciones a las copias secundarias. En general, si la copia primaria (o su nodo) falla, se elige entre las copias secundarias una que pase a desempeñar dicho papel.

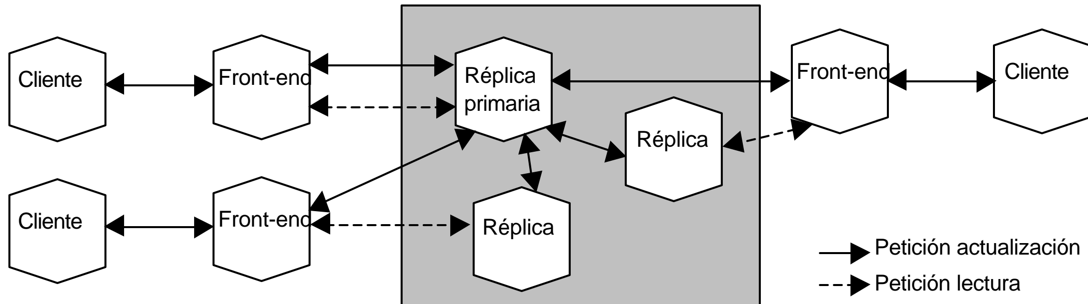


Figura 13.10. Esquema de replicación de copia primaria.

En un esquema de **replicación de objetos iguales** no se distinguen objetos primarios y secundarios. Todas las peticiones, modifiquen o no el estado del objeto, pueden ser servidas por cualquier réplica. No obstante, se necesita la cooperación de todas las réplicas con el fin de procesar una petición.

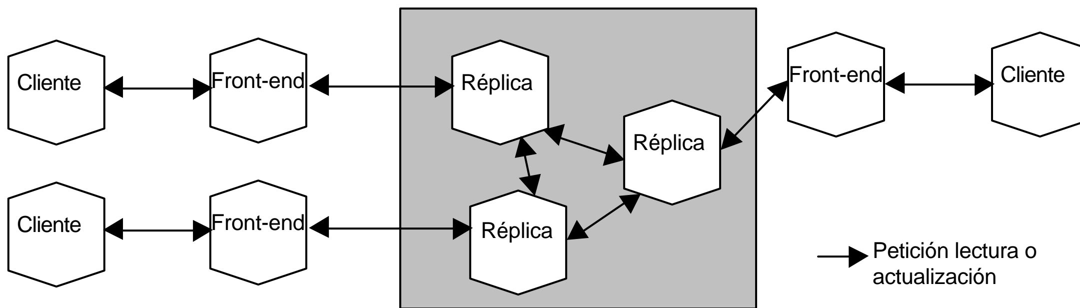


Figura 13.11. Esquema de replicación de objetos iguales.

13.4.1 Introducción de objetos replicados en el sistema integral

Como ya se ha comentado, **el modelo de objetos soportado por las máquinas abstractas no incluye el concepto de objeto replicado**. Se cumple de esta manera con uno de los principales objetivos de diseño de la máquina abstracta que es la simplicidad del modelo. Por otro lado, y como también se ha apuntado ya, la replicación de objetos introduce importantes beneficios en un sistema integral distribuido: mejoras en la accesibilidad y el rendimiento y tolerancia a fallos.

La extensión de un modelo de objetos distribuidos de única copia a un modelo de objetos replicados vuelve a pasar por la característica reflectiva de la máquina abstracta. Los meta-objetos del sistema operativo serán los encargados de proporcionar al usuario la ilusión de un modelo de objetos de copia única con las ventajas de la replicación.

El primer servicio que se ve afectado por la introducción de la replicación en el sistema integral es el servicio de nombrado. **El servicio de nombrado debe permitir la existencia de diferentes identidades de objetos para el mismo nombre simbólico**. Ante la petición de un objeto cliente, el servicio de nombrado devolverá una referencia a la réplica más cercana (entendiendo cercana en el sentido más amplio de la palabra) al cliente. El objeto cliente utilizará dicha réplica como si fuera la única copia existente del objeto.

En segundo lugar, **el servicio de replicación debe estar de acuerdo con la facilidad de migración**. No sería lógico que en un mismo nodo coexistiesen varias réplicas del mismo objeto cuando otros nodos no dispusiesen de ninguna aun cuando en

él estuviera ubicado alguno de sus objetos clientes. Las distintas políticas de migración deben intentar mantener una relación uno a uno, como máximo, entre nodos y réplicas de cada uno de los objetos replicados.

Finalmente, es necesario **modificar el mecanismo de invocación** de métodos. En el capítulo anterior se describió cómo la reflectividad de la máquina abstracta permite modificar el mecanismo de invocación para introducir una facilidad de invocación remota. En el apartado dedicado a las transacciones, en este mismo capítulo, se describió cómo modificar el mecanismo de invocación para introducir la gestión de transacciones. De una manera similar, se introducirá la replicación.

En la figura siguiente se muestra un esquema de la implantación de la replicación de copia primaria en el sistema integral a través del uso de los meta-objetos emisor y receptor. Los meta-objetos emisor de los objetos cliente interceptan todas las invocaciones a cualquiera de las réplicas del objeto servidor. Si la operación es de lectura, la invocación es desviada a la réplica asociada al objeto cliente (la más próxima) y recibida por su meta-objeto receptor. Si se trata de una operación de actualización, la invocación es desviada al meta-objeto receptor de la réplica primaria. Finalmente, el meta-objeto receptor de la copia primaria se encarga de propagar las modificaciones de la misma al resto de las réplicas.

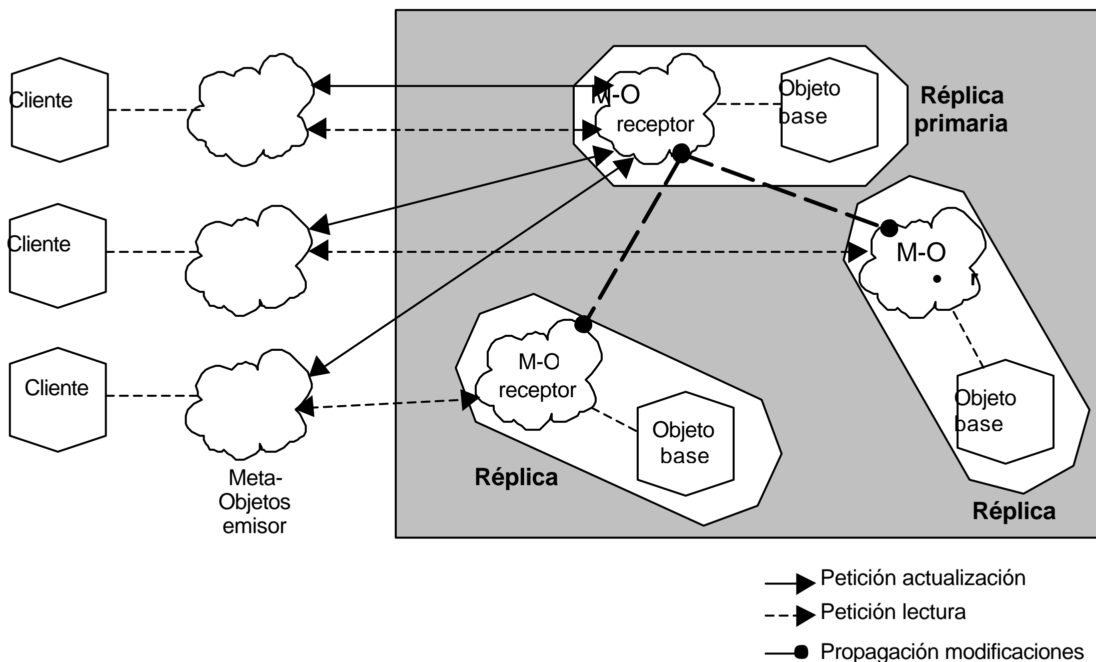


Figura 13.12. Esquema de replicación de copia primaria para el sistema integral.

13.5 Recolección de basura distribuida

Lenguajes de programación como Pascal, que utilizan asignación dinámica de memoria dejan en manos del programador la liberación de la memoria asignada. En estos lenguajes el programador tiene que asignar y liberar memoria explícitamente, a través de llamadas a rutinas de gestión de memoria dinámica. Una gestión de memoria explícita como la descrita no es deseable debido a varias razones:

- Errores [Rov86]. Los errores más comunes incluyen la no liberación de memoria que ya no está en uso (provocando, en algún caso, escasez de memoria disponible) y la liberación de memoria que todavía está en uso (dejando punteros que apuntan a memoria libre, y que causan a menudo errores difíciles de localizar).
- Transparencia en la gestión de memoria. La gestión de memoria es un tema de implementación y como tal, debería ser transparente al programador [PS75].
- Libertad de implementación. La ocultación de la gestión de la memoria al programador da al implementador la libertad de elegir entre diferentes alternativas de implementación.
- Liberación de recursos del sistema. Algunos recursos del sistema tienen que ser recolectados y no pueden serlo bajo petición del programador porque su existencia les es transparente.

Una gestión de memoria que utiliza asignación y **recolección de basura** (*garbage collection*) automáticas tendrá en cuenta todas estas cuestiones, pero introduce un conjunto de problemas nuevos.

La recolección de basura no debería degradar de manera significativa el rendimiento del sistema. El rendimiento se puede ver afectado de dos maneras. En primer lugar, el rendimiento global se puede degradar si se utiliza un tiempo excesivo en recolectar basura. En segundo lugar, el tiempo de respuesta puede verse afectado porque la aplicación se retrasa realizando acciones de recolección de basura (por ejemplo, los recolectores clásicos del tipo *mark-and-sweep* provocan pausas largas y repentinas).

En sistemas orientados a objetos tradicionales, la recolección de basura es necesaria. Por ejemplo, debido a la encapsulación, un programador no tiene acceso directo (o incluso no conoce) a los subobjetos de un objeto, y no los puede liberar cuando libera el objeto.

Se describen a continuación dos algoritmos de recolección de basura para ilustrar una posterior implantación en el sistema integral: los algoritmos de cuenta de referencia y de marcado y rastreo.

13.5.1 Algoritmos de recolección de basura

Los **algoritmos de cuenta de referencia** operan contando el número de referencias directas que existe para cada objeto. El número es almacenado con el objeto y es denominado su **cuenta de referencia**. Es incrementado cuando se crea una nueva referencia al objeto y se decrementa cuando se destruye una referencia al mismo. Si la cuenta de referencia llega a tomar el valor cero, el objeto es inalcanzable y puede ser reclamado inmediatamente para ser recolectado. Cuando se recolecta un objeto, el recolector destruye todas las referencias almacenadas en el objeto, lo cual puede provocar a su vez que la cuenta de referencia de algún otro objeto llegue también a cero, siendo reclamado este objeto de manera recursiva por el recolector.

Los algoritmos de cuenta de referencia tienen la **ventaja** de que son incrementales: los objetos son reclamados inmediatamente para ser recolectados y la sobrecarga de la recolección se distribuye de manera uniforme entre las computaciones que se están realizando.

Como **desventajas**, citar la imposibilidad de recolectar ciclos (objetos que se referencian mutuamente), la necesidad de incrementar el espacio de almacenamiento

para los objetos, con el fin de registrar la cuenta de referencia, la sobrecarga en las operaciones con referencias (una asignación entre referencias necesita una operación de incremento y otra de decremento de las cuentas de referencia de los objetos) y los problemas que presenta para un sistema distribuido. En este último caso, los problemas se presentan por la necesidad de actualizar un valor (la cuenta de referencia) que se encontrará allá donde se encuentre el objeto en el sistema distribuido cuando las referencias al objeto que se crean y se destruyen pueden estar, a su vez, en cualquier otro nodo.

Por otro lado, los algoritmos de **marcado y rastreo** (*mark-and-sweep*) trabajan, en una primera fase, marcando todos los objetos que son accesibles desde un conjunto de objetos raíz (objetos distinguidos como accesibles aunque no haya referencias a ellos) y, en una segunda fase, reclamando para ser recolectados todos aquellos objetos que han quedado sin marcar.

De una manera más concreta, en la fase de marcado se marcan todos los objetos alcanzables siguiendo las referencia a objetos presenten en los objetos raíz. A continuación, y siguiendo las referencias a objetos presenten en los objetos marcados, se marca un segundo conjunto de objetos, y así sucesivamente. Todos los objetos no marcados serán recolectados. En la fase de rastreo, el recolector de basura rastrea el área de almacenamiento y reclama todos los objetos no marcados.

La **ventaja** principal de estos algoritmos es que recolectan todos los objetos susceptibles de serlo, incluidos los objetos que forman estructuras circulares.

Las **desventajas** vuelven a ser varias. El algoritmo tradicional no es concurrente con las tareas de procesamiento ordinarias, de tal forma que todo el procesamiento se tiene que detener mientras se realiza la recolección. Además, no son incrementales, puesto que la recolección se hace toda de una vez. Finalmente, introducen un problema de rendimiento, dado que es preciso visitar todos los objetos alcanzables, visitas que involucran todos los nodos del sistema distribuido.

13.5.2 Recolección de basura en el sistema integral

La recolección de basura se presenta, en un principio, como una operación necesaria pero con un gran coste en un entorno distribuido. El sistema integral orientado a objetos tiene que dar solución a los objetos no alcanzables (y por tanto, inútiles) sin introducir problemas graves de disminución del rendimiento.

En [ADA+99] se describe la forma de eliminar prácticamente por completo la necesidad de la recolección de basura en un sistema integral orientado a objetos. Las claves se encuentran en la propiedad de **persistencia completa** del sistema y la utilización de un espacio de **almacenamiento terciario**.

La persistencia completa (ver capítulo siguiente) se utiliza para convertir el área de instancias de las máquinas abstractas del sistema integral en un área de instancias virtual, con la utilización de un almacenamiento secundario en el que también pueden residir los objetos. El sistema de persistencia utiliza técnicas tradicionales de memoria virtual para mover objetos entre el almacenamiento principal (las área de instancias de las máquinas) y el persistente.

Esta arquitectura permite, en primera instancia, evitar la recolección de basura en las áreas de instancias. Un objeto no alcanzable que resida en el área de instancias de una máquina será llevado al almacenamiento de persistencia más tarde o más temprano,

dado que su espacio será reclamado para objetos que estén en uso (que se invoquen con cierta frecuencia).

Dado que el almacenamiento de persistencia funciona como una extensión de las áreas de instancias reales, se pueden encontrar en él tanto objetos alcanzables como no alcanzables. Se añade un tercer nivel de almacenamiento, en el que se colocarán todos los objetos que ocupaban espacio en el almacenamiento secundario y que no hubieran sido utilizados durante un largo periodo de tiempo. Los objetos de este almacenamiento terciario podrían abandonarlo en el momento en que fuesen referenciados. Solamente los objetos inalcanzables permanecerían indefinidamente en él, haciendo innecesaria la recolección de basura.

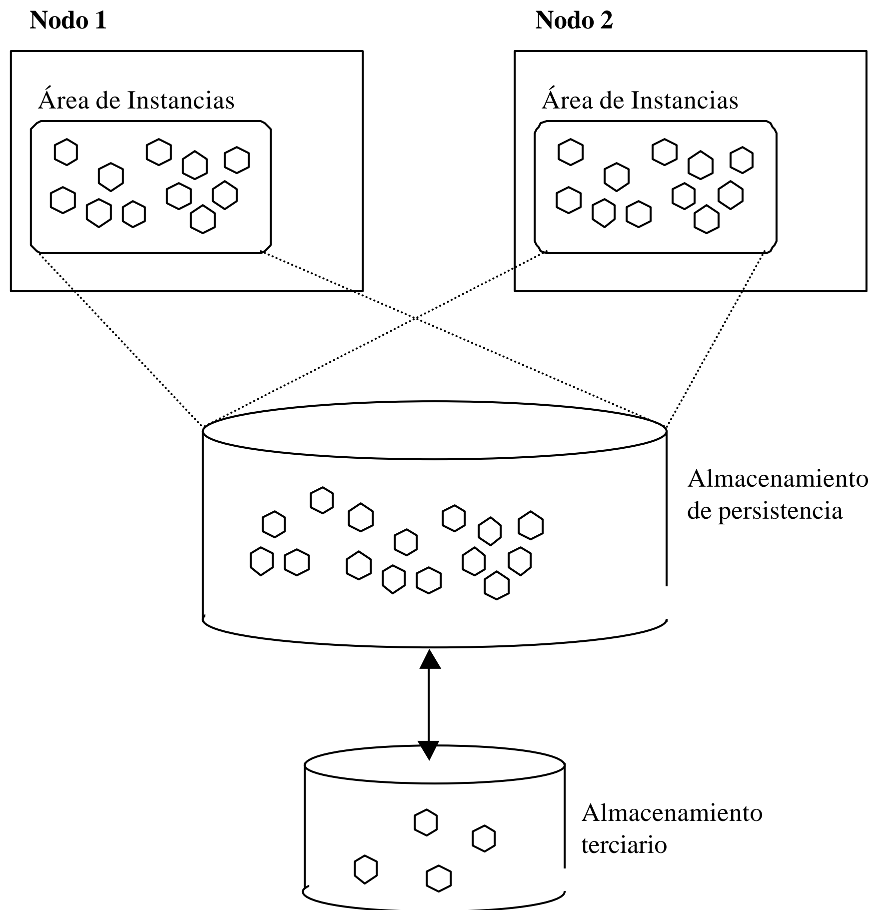


Figura 13.13. Utilización de un almacenamiento terciario.

Cabe preguntarse si son aceptables los costes de esta arquitectura en contraposición a la utilización de una recolección de basura tradicional. En primer lugar, la mayor parte de los objetos serán eliminados implícitamente, sin necesidad de control por parte del programador. Por ejemplo, el lenguaje de bajo nivel de la máquina abstracta puede eliminar los objetos locales a un método que no son devueltos ni pasados a otros métodos. En segundo lugar, el abaratamiento del almacenamiento permite su utilización con el fin de no necesitar realizar recolección de basura. Finalmente, si se decide realizar recolección de basura, su coste se reduce sensiblemente, dado que los objetos candidatos a ser recolectados está identificados de antemano y se corresponden con aquellos objetos que se encuentran en el almacenamiento terciario.

13.6 Resumen

En general, un sistema operativo distribuido orientado a objetos no proporciona únicamente servicios de invocación remota y migración de objetos. El diseñador del sistema operativo incluye servicios adicionales que permiten, a su vez, dar un mejor servicio a los usuarios.

El núcleo del sistema de distribución de objetos (Agra) se diseñó con el objetivo de proporcionar la funcionalidad mínima y estar preparado para proporcionar una mayor funcionalidad basándose en la anterior. De ahí, que los servicios descritos en este capítulo se consideren como adicionales a los básicos descritos en los capítulos anteriores.

El servicio de nombrado permite obtener referencias a objetos que son habitualmente utilizados por una gran cantidad de objetos a partir de un nombre simbólico. Los nombres simbólicos toman, en general, la forma de los nombres de ficheros de los sistemas operativos convencionales.

El servicio de transacciones proporciona un grado superior de consistencia de los datos a la ya proporcionada por los objetos. De todas formas, hacer que todas las invocaciones a objetos sean tratadas como transacciones tiene un coste muy elevado, por lo que es necesario introducir facilidades para que el usuario especifique los límites de una transacción.

El servicio de replicación permite mejorar el rendimiento del sistema, evitando aquellos componentes centralizados muy demandados, y su tolerancia a fallos, si cualquiera de las réplicas es capaz de atender cualquier tipo de petición. Dado que el modelo de objetos soportado por la máquina abstracta no ofrece la abstracción de objeto replicado, es necesario ofrecerla en el sistema operativo.

Todos los servicios anteriores se sustentan sobre la reflectividad del sistema. De esta forma, el diseñador del sistema operativo decide qué servicios incluir para una implantación concreta, sin que la máquina abstracta subyacente ni el usuario final sean conscientes de ello, excepto en los beneficios propios proporcionados por el servicio.

Finalmente, la recolección de basura distribuida, puede ser evitada en gran medida gracias a la utilización, en primer lugar, de un sistema de persistencia completa (ver capítulo siguiente) y, en segundo lugar, de un espacio de almacenamiento terciario. Los costes asociados a los algoritmos de recolección de basura tradicionales son prácticamente eliminados.

CAPÍTULO 14 RELACIÓN DEL SUBSISTEMA DE DISTRIBUCIÓN CON OTROS SUBSISTEMAS

14.1 Introducción

Aunque esta Tesis trata de la Distribución de Objetos para un sistema operativo que se ejecuta en un Sistema Integral Orientado a Objetos, es necesario conocer de qué manera se relaciona con otros componentes que también pueden estar presentes.

La característica de persistencia ya ha sido considerada como de las integrantes del modelo de objetos a implantar en un sistema de soporte de objetos. Su utilización, conjuntamente con la distribución, es una de las bases de la implantación de los servicios de transacciones de objetos, tal y como se comentó en el capítulo anterior.

Por su parte, la seguridad es un elemento clave en cualquier sistema operativo moderno. Es preciso proteger todos los elementos del sistema contra usos no adecuados, ya sea por error o intencionadamente. Como ya se ha repetido muchas veces a lo largo de este trabajo, el sistema de protección no puede romper la uniformidad en la orientación a objetos que hasta este punto se ha venido manteniendo.

14.2 Subsistema de persistencia

La propiedad de persistencia de los objetos del sistema integral ha aparecido en repetidas ocasiones y se ha descrito como una propiedad básica. A continuación se describe la forma en que se introduce la persistencia en el sistema integral. Una descripción completa se puede encontrar en [Álv98, Ort97].

14.2.1 Elementos básicos de diseño del sistema de persistencia

En esencia se trata de proporcionar la abstracción de un **espacio de objetos potencialmente infinito** en el que coexisten simultáneamente e indefinidamente todos los objetos del sistema, hasta que ya no son necesitados. Podría considerarse como una memoria virtual persistente para objetos. El usuario simplemente trabaja con los objetos en ese espacio virtual.

El sistema de persistencia deberá proporcionar con los recursos existentes en el entorno (a partir de la máquina abstracta orientada a objetos) la abstracción anterior.

Los puntos básicos del sistema de persistencia del sistema integral se refieren a continuación.

14.2.1.1 Persistencia completa

Absolutamente todos los objetos creados en el sistema son persistentes por definición. Esto incluye también a los objetos del sistema operativo, que deben tener la misma categoría que los objetos de usuario. No es necesario almacenar y recuperar explícitamente los objetos, el sistema lo realiza de manera transparente. Sólo existe un único conjunto de operaciones para manipular los objetos.

14.2.1.2 Estabilidad y elasticidad

El espacio virtual persistente de objetos debe proporcionar las propiedades de estabilidad y elasticidad. La estabilidad es la capacidad de un sistema para registrar su estado de manera consistente en un medio seguro, de tal forma que el funcionamiento podría reanudarse a partir de ese punto en el futuro. Un sistema es elástico si puede reanudarse el funcionamiento con seguridad después de una caída inesperada del mismo, como por ejemplo, un fallo de alimentación.

14.2.1.3 Encapsulamiento de la computación

Se propone un modelo de objetos que además encapsule como parte de su estado la computación, que tendrá que ser perfilado por la parte de concurrencia del sistema operativo.

14.2.1.4 Identificador uniforme único

Se utilizará el identificador único de objetos de la máquina para referenciar los objetos, independientemente de su localización física. Este identificador es usado tanto por el usuario como por el sistema internamente para referenciar el objeto en memoria principal y en almacenamiento persistente. Esto tiene mucha similitud con el modelo de sistema operativo de espacio de direcciones virtual único [VRH93, SMF96], jugando el papel de las direcciones virtuales los identificadores de los objetos. Este espacio único se extiende a la memoria secundaria, formando un sistema de almacenamiento con sólo un nivel (*single level store* [KEL+62]).

14.2.1.5 Ventajas

La utilización de un esquema como el que se ha propuesto conlleva unas ventajas, de las cuales destacan las siguientes:

Permanencia automática

Una vez que se crea un objeto en el sistema, este permanece el tiempo necesario hasta que no sea necesitado más. No es necesario preocuparse de almacenar el objeto en almacenamiento secundario si se desea que no se pierda su información. Esto facilita la programación.

Abstracción única de memoria. Uniformidad

La única abstracción necesaria para la memoria es el objeto. El sistema de manera transparente conserva la información del objeto siempre. Al eliminar la dualidad de la abstracción entre memoria de largo plazo y de corto plazo, y sustituirlas por una única abstracción de espacio virtual persistente de objetos se facilita la labor de los programadores. Abstracciones como fichero ya no son necesarias (aunque podrían implementarse como objetos normales, si se desea). Simplemente se trabaja con un único paradigma, que es el de la orientación a objetos.

Permanencia de la computación

Al encapsular el objeto también el procesamiento, no sólo se conservan los datos, si no también la computación [KV92, TYT92]. Se mantiene totalmente el estado: datos y proceso. Esta característica se suele considerar ya parte integrante de un modelo de objetos [Boo94].

Memoria virtual persistente distribuida

Se crea fácilmente una verdadera memoria virtual distribuida de objetos al utilizar un identificador único de objetos. Simplemente basta con utilizar un mecanismo que haga que este identificador sea diferente para todos los objetos, independientemente de la máquina en que se crearon. Se utilizará este identificador único en conjunción con el mecanismo de distribución para localizar en todo el sistema distribuido un objeto dado.

14.2.2 Implementación de la persistencia en el sistema integral

Las referencias pueden considerarse como punteros a otros objetos, aunque se diferencian de los punteros convencionales por el hecho de que siempre son válidos, al utilizarse el identificador único del objeto.

El elemento fundamental de la máquina es el área de instancias, que almacena los objetos. Este área es el equivalente a la memoria principal de los sistemas convencionales.

14.2.2.1 Área de instancias virtual persistente

En esencia, se implementa la persistencia como una extensión del área de instancias. Es decir, lograr la ilusión de un **área de instancias virtual**[15] (**memoria virtual**), utilizando para ello un almacenamiento secundario para hacer persistir los objetos, guardándolos cuando no estén (o no quepan) en el área de instancias. Todo ello de manera totalmente transparente para el resto del sistema.

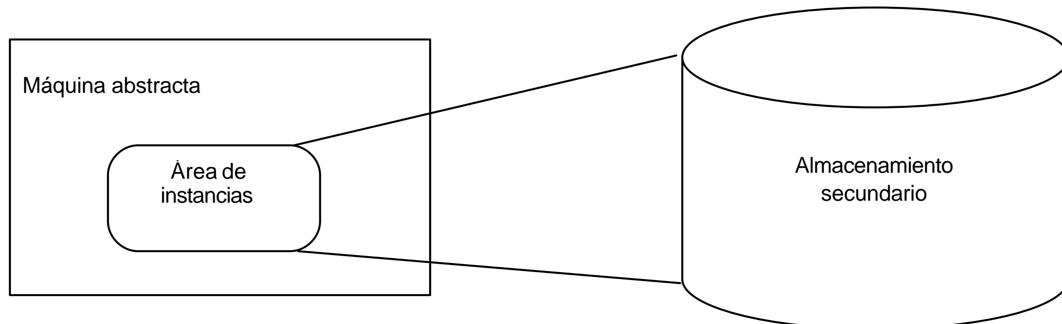


Figura 14.1. Área de instancias virtual.

Se unifican los principios de memoria virtual tradicionales de los sistemas operativos [Sta98] con los de persistencia. El sistema proporciona un espacio de memoria único en el que residen todos los objetos (persisten), virtualmente infinito.

14.2.2.2 Identificador uniforme de objetos igual al identificador de la máquina

Como identificador del objeto en almacenamiento persistente se utilizará de manera uniforme el propio identificador del objeto dentro de la máquina. Es decir, se usa en todo momento un único identificador del objeto, que siempre es válido en cualquier situación en la que se encuentre el objeto: tanto en almacenamiento persistente como en el área de instancias.

[15] Hay que recalcar que, dentro de la semántica de un objeto, un elemento fundamental es la clase a la que pertenece. Por tanto, aunque se haga referencia únicamente a los objetos, se entiende implícitamente que los mismos mecanismos se aplican para hacer persistir las clases de los objetos. Es decir, la persistencia se aplica de manera análoga al área de clases.

14.2.2.3 Mecanismo de envío de mensajes y activación del sistema operativo por reflexión explícita

Cuando se envía un mensaje a un objeto (en general cuando se necesite acceder a un objeto por alguna razón), se proporciona a la máquina una referencia al mismo (identificador del objeto). La máquina usa este identificador para localizar el objeto en el área de instancias. Si el objeto no está en esta área debe generarse una excepción o mensaje que permita intervenir al sistema operativo. Mediante la reflectividad se hace una reflexión explícita que ceder el control a un objeto del sistema operativo.

14.2.2.4 Objeto "paginador"

Se activará un objeto colocado al efecto por el sistema operativo. La función de este objeto es precisamente ocuparse del trasiego de los objetos entre el área de instancias y el almacenamiento persistente. Este objeto y el mecanismo en general son similares al concepto del paginador externo para memoria virtual de sistemas operativos como Mach [ABB+86] y, por tanto, se le denominará objeto "**paginador**".

14.2.2.5 Carga de objetos

Para localizar el objeto en almacenamiento persistente se utilizará el identificador del objeto proporcionado por la referencia usada en el envío del mensaje.

Una vez localizado el objeto, se debe colocar en el área de instancias, reconstruyendo en efecto el estado completo del mismo. Para ello se invocarán métodos adecuados en la meta-interfaz del objeto que representa reflectivamente el área de instancias.

Al finalizar el trabajo del objeto "paginador", debe regresar el control a la máquina para que continúe con su funcionamiento normal, al estar ya el objeto en el área de instancias.

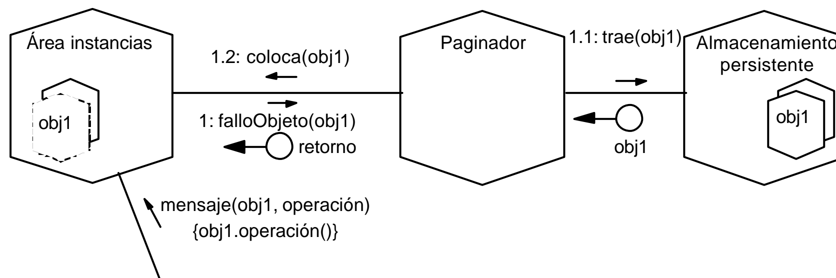


Figura 14.2. Carga de un objeto en el área de instancias.

14.2.2.6 Reemplazamiento

En caso de no existir sitio en el área de referencias para situar un objeto, bien en la creación de un nuevo objeto, bien por necesitarse la carga del mismo por el motivo anterior, debe liberarse espacio en la misma. Por un mecanismo similar al anterior se activará el objeto paginador y seleccionará uno o varios objetos que llevará al almacenamiento persistente, registrando su estado y liberando así el espacio necesario. Para seleccionar los objetos a reemplazar puede utilizarse otro objeto que proporcione la estrategia de reemplazo.

Hay que destacar que dado que el objeto encapsula la computación, también se conserva el estado de la misma.

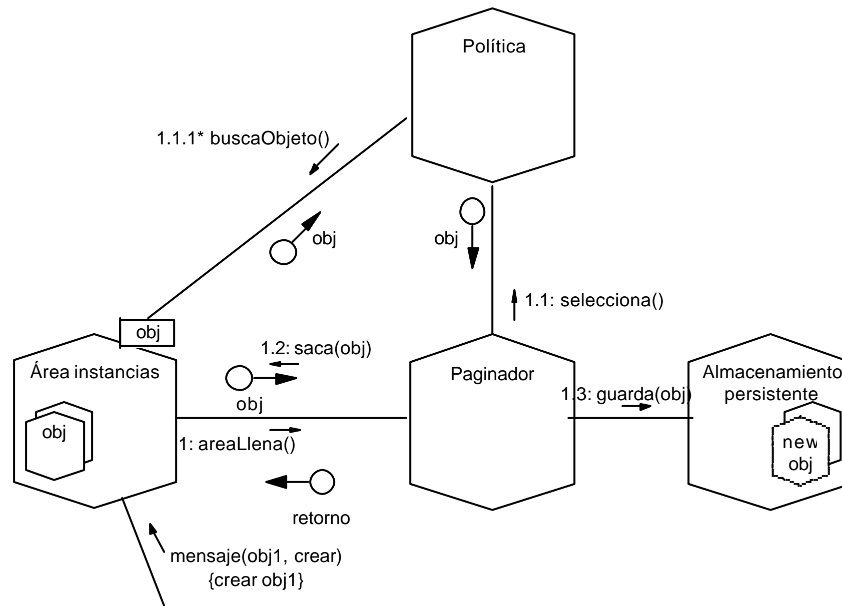


Figura 14.3. Reemplazamiento de un objeto en el área de instancias.

14.2.2.7 Manipulación interna

Es necesario habilitar un mecanismo que permita a este objeto del sistema manipular el área de instancias, para lo cual se aprovecha la arquitectura reflectiva de la máquina. Algo similar debe hacerse para poder registrar el estado de los objetos, al tener que acceder a su representación interna, bien por el método anterior, bien definiendo unas operaciones primitivas disponibles en todo objeto que lo permitan.

14.2.3 Relación con la distribución

La combinación de los mecanismos de distribución y persistencia permite obtener un **mundo de objetos** virtualmente infinito en el espacio (distribución) y en el tiempo (persistencia), en el que un conjunto de objetos coopera intercambiando mensajes independientemente de su localización y donde los objetos residen hasta que ya no son necesitados.

La relación principal entre los dos subsistemas se establece en la invocación de métodos, y en particular, en la operación de localización del objeto invocado.

Las referencias de los objetos no contienen información de localización y, en un momento dado, el objeto invocado puede encontrarse en alguna de las situaciones siguientes:

- Existe copia persistente del objeto pero no existe copia volátil del mismo en ningún nodo del sistema integral.
- Existe copia volátil del objeto en el mismo nodo en el que reside el objeto que invoca.
- Existe copia volátil del objeto en un nodo distinto del nodo en el que reside el objeto que invoca.
- El objeto invocado ya no existe.

Por tanto, una operación de localización de un objeto puede llegar a involucrar a los subsistemas de distribución y persistencia.

Si **a la hora de invocar un objeto**, el sistema de localización se encuentra con que no existe copia volátil del mismo y sí existe copia persistente, se habrá de solicitar al sistema de persistencia que active (cree una copia volátil) el objeto invocado. Dado que los mecanismos de persistencia y distribución son independientes, no se tiene control sobre el nodo del sistema integral en el que se va a ubicar la copia volátil del objeto. En otras palabras, el control sobre la ubicación del objeto activado dependerá completamente de las políticas del mecanismo de persistencia. Por ejemplo, la política de activación de objetos puede establecer que los objetos se activen en el nodo desde el que son reclamados (invocados). Una vez obtenida la copia volátil del objeto, el proceso de invocación puede continuar, enviándose un mensaje al nodo en el que se ha activado conteniendo los parámetros de la invocación.

A efectos de la **migración de objetos**, la persistencia no introduce ningún tipo de distorsión. Solamente las copias volátiles de los objetos son consideradas a efectos de la migración. Aquellos objetos que se encuentran en el almacenamiento de persistencia y que están relacionados con objetos que migran, se activarán con toda probabilidad (si la política de activación está bien diseñada) en el nodo destino de dicha migración cuando sean invocados.

Hay que destacar la **ortogonalidad** existente entre los subsistemas de persistencia y distribución, que pueden existir independientemente de la existencia del otro. Dicha independencia se consigue gracias a la posibilidad de modificar el comportamiento por defecto de diferentes mecanismos de la máquina abstracta a través del uso de la reflectividad.

14.3 Subsistema de protección

El subsistema de protección se convierte en el núcleo de seguridad para el sistema integral. Un buen mecanismo de protección deberá ser lo suficientemente flexible para que puedan implementarse sobre él diversas políticas de seguridad. A pesar de que las políticas de seguridad tienen su importancia, no se tratan en esta tesis. Para una información completa sobre el mecanismo de protección y las políticas de seguridad, consultar [Día2000].

14.3.1 Requisitos de protección

A continuación se establecen los requisitos principales que se van a exigir al mecanismo de protección, en consonancia con las expectativas que impone la utilización de un sistema integral orientado a objetos.

14.3.1.1 Flexibilidad

Un buen diseño de un mecanismo de protección debe ser flexible. Esto significa que este mecanismo debe ser capaz de servir como base a diversas políticas de seguridad. El mecanismo se encargará de comprobar la existencia de permisos frente a una determinada petición de un recurso. A un nivel superior se encuentra la política que determina los permisos que se conceden en el sistema, la propagación y restricción de los mismos, etc.

14.3.1.2 Movilidad

El sistema de seguridad debe tener en cuenta la movilidad de los objetos. En un SIOO los objetos se distribuyen entre un conjunto de máquinas y pueden moverse sin restricciones de una a otra, lo que supone un peligro de seguridad adicional. Como se explicó en el capítulo 5 sobre el SIOO, los objetos son autocontenidos y por tanto cada objeto lleva consigo toda la información relativa necesaria para su ejecución en cualquier punto del sistema. El mecanismo de protección deberá respetar este modelo de diseño integrando en él los aspectos necesarios para garantizar la protección sin imponer restricciones a la movilidad.

14.3.1.3 Uniformidad en la Orientación a Objetos

El mecanismo de protección debe respetar la uniformidad en la OO del sistema. Es decir, no deberá introducir abstracciones adicionales que rompan el modelo de objetos del sistema.

14.3.1.4 Homogeneidad

El mecanismo de protección debe ser homogéneo, es decir, debe proteger a todos los objetos de igual manera. Para que el sistema en su conjunto sea conceptualmente sencillo de entender para el usuario, al igual que el modelo de objetos del sistema es uniforme y homogéneo, esa misma uniformidad y homogeneidad se desea para el mecanismo de protección.

Por tanto, la protección debe realizarse en el ámbito de objetos de cualquier granularidad, dado que en un SIOO todos los objetos se consideran como entidades del mismo nivel, independientemente de su granularidad. Esto es, cualquier invocación a un método de un objeto, independientemente del tamaño que éste tenga, debe implicar una comprobación de permisos, pues la protección se debe prestar a todos y cada uno de los objetos existentes en el sistema. No se debe hacer distinción entre tipos de objetos puesto que se consideran todos los objetos iguales, y por tanto también lo deben ser en cuanto a protección.

14.3.1.5 Protección de granularidad fina

Por otro lado, la protección debe extenderse al nivel de los métodos individuales de los objetos. Es decir, debe poder especificarse los permisos que cualquier objeto individual tiene sobre cualquier otro objeto. Este requisito podría implicar una mayor sobrecarga producida por la protección, al necesitar comprobaciones en todas las operaciones del sistema. Sin embargo, también supone un esquema de protección más completo, ya que garantiza una mediación total, al controlar el funcionamiento hasta en los componentes más elementales (objetos) del sistema. Además, esta protección de las operaciones más elementales del sistema, permite un control mucho más detallado y versátil.

14.3.2 Modelo de protección

El modelo básico de protección del sistema integral está basado en capacidades. En concreto, se ha diseñado un tipo de capacidades denominadas **capacidades orientadas a objetos**, que aprovechan parte de las ventajas que proporciona el uso de una máquina abstracta orientada a objetos con su lenguaje ensamblador orientado a objetos. El propio diseño del repertorio de instrucciones del lenguaje es seguro: garantiza que la única manera de manipular los objetos (datos, incluidas las propias capacidades) es a través de las instrucciones definidas al efecto. Es imposible, por tanto, acceder a la representación

física de los objetos y cambiar de manera arbitraria estos. Es decir la propia máquina abstracta evita la posibilidad de falsificación de las capacidades.

La idea consiste en **implementar las capacidades integrando la información de protección con la referencia al objeto** que existe en la máquina abstracta, añadiendo a ésta los permisos que el poseedor de la referencia tendrá sobre el objeto al que se hace referencia. A partir de este momento, las referencias de la máquina siempre incorporan la información de protección sobre el objeto al que apuntan. Es decir, las referencias se convierten en capacidades[16]. En este sentido, se tiene un sistema basado en capacidades puras, en las que las capacidades combinan denominación (referencia a un objeto) y protección, y esta es la única manera de acceder a los objetos.

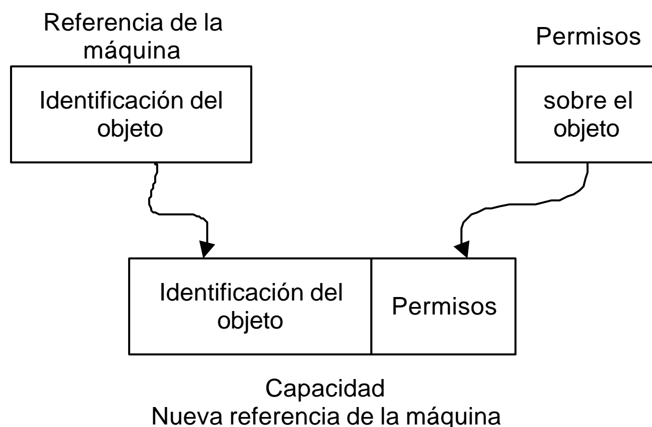


Figura 14.4. Estructura interna de una capacidad.

14.3.2.1 Capacidades con número de permisos variable

En un SIOO los objetos tienen la semántica y el nivel de abstracción tradicional de las metodologías. Por tanto pueden tener un conjunto arbitrario de operaciones, tanto en número como en tipo. La homogeneidad y la protección de granularidad fina que necesita un SIOO requiere que todos los métodos de cualquier objeto sean protegidos de forma individual.

Parece evidente que un mecanismo de protección flexible y completo consiste en un tipo de capacidades que dé soporte a un número variable de permisos (tantos como métodos tenga el objeto). Una ventaja del SIOO es que las referencias son abstracciones de alto nivel del sistema, y como tales son manejadas por las aplicaciones. Estas desconocen la implementación física interna de las capacidades y no se hace ninguna suposición sobre cómo puede ser ésta (por ejemplo, que tenga un tamaño determinado). Esto permite que sea posible implementar esta abstracción con un número variable de permisos sin afectar en absoluto al resto del sistema.

Se utiliza, por tanto, un mecanismo de protección al nivel de todas las operaciones de cualquier objeto. El número de permisos almacenados en la capacidad será variable en función de los métodos que tenga el objeto al que apunte la referencia/capacidad.

---

[16] En lo sucesivo se utilizarán las denominaciones de referencia o capacidad indistintamente, ya que describen el mismo concepto

14.3.2.2 Salto de la protección

En un SIOO como el planteado, con objetos de cualquier granularidad, muchos de estos objetos van a formar parte de objetos locales a otros de mayor granularidad. Estos objetos normalmente sólo serán utilizados por sus propietarios (objeto que los creó), y no van a ser accedidos por otros objetos que su creador. Es decir, estos objetos no van a ser compartidos. En estos casos la necesidad de protección de los métodos de estos objetos no suele ser necesaria, y puede dejarse que se tenga permiso de acceso a todos sus métodos.

En los casos en los que la capacidad tiene todos los permisos activos, y por tanto el mecanismo de protección permitirá realizar cualquier operación, se utiliza un permiso especial que indica "Salto de protección". En estos casos el mecanismo de protección no comprobaría los permisos, si no que directamente pasaría a la ejecución del método en cuestión.

14.3.3 Implantación en la máquina abstracta

La comprobación de la protección se implanta como parte integrante de la propia máquina abstracta del sistema integral.

Dado que la máquina ya tiene integrado un mecanismo para realizar el paso de mensajes entre los objetos a través de referencias, es sencillo introducir la comprobación de permisos en el propio paso de mensajes. Así, cuando se envía un mensaje a un objeto, la máquina simplemente debe hacer un paso adicional: comprobar que dentro de los permisos que están en la capacidad se encuentra el necesario para invocar el método solicitado. Si es así se procede normalmente, en caso contrario se devuelve una excepción de protección al objeto que intentó la llamada.

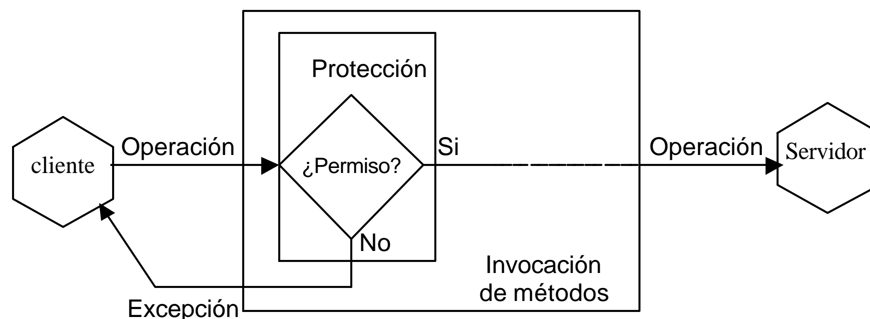


Figura 14.5. Integración del mecanismo de protección junto con la invocación de métodos, dentro de la máquina abstracta

Con esto se consigue mantener la uniformidad del modelo de objetos existente, puesto que externamente no se cambia la funcionalidad anterior del sistema. Por otro lado se consigue de una manera sencilla e integrada en el corazón del sistema: todas las operaciones en el sistema (pasos de mensajes) están sujetas al mecanismo de protección y además es imposible saltarse el mecanismo ya que es parte del nivel más bajo del sistema.

14.3.4 Modo de operación del mecanismo

En este apartado, se resume cómo funciona el modelo de protección elegido y se integra dentro de la máquina de forma natural y proporciona una protección no esquivable, a la vez que totalmente flexible.

El mecanismo de protección funcionará de la siguiente manera:

1. Para poder realizar una operación sobre un objeto es necesario tener su capacidad. La consecución de ésta es transparente al modelo de objetos puesto que está incluida en la referencia. En un lenguaje de programación OO, para poder operar sobre un objeto es necesario contar con una referencia a éste.
2. Existen dos modos de obtener una capacidad: obteniéndola al crear un objeto o recibiéndola como parámetro en una invocación del método. Este mecanismo es totalmente transparente, puesto que coincide con la forma de obtener referencias en un modelo de objetos.
3. En una capacidad aparece información acerca de los permisos existentes sobre cada uno de los métodos del objeto. Si la capacidad otorga permisos sobre todos los métodos, entonces el permiso especial de “salto de protección” estará activo, en caso contrario está inactivo. Esta información forma parte de los atributos de la referencia al objeto.
4. Cuando se realiza una invocación a un método, la máquina la resuelve, localizando el objeto, la clase a la que pertenece y el método invocado. Antes de transferir la ejecución al método del objeto invocado, la máquina comprueba el estado del permiso de “salto de protección” localizado en la referencia/capacidad. Si éste está activo se pasará a realizar la invocación. Si no lo está, la máquina comprueba el estado del permiso correspondiente al método invocado, situado en la referencia/capacidad. Si éste está activo, se efectúa la invocación. En caso contrario se lanza una excepción de protección.
5. Un objeto A que posee una capacidad sobre otro B puede pasársela a un tercero C, para que C tenga también capacidad sobre B. El modo de pasar capacidades será como parámetro en una invocación a uno de sus métodos. Este mecanismo es transparente puesto que funciona como las referencias en un modelo de objetos.
6. Un objeto nunca podrá modificar ni aumentar el número de permisos de una capacidad por sí mismo (falsificación). Puesto que los permisos son atributos privados de las capacidades, no existe modo alguno de que sean manipulados, excepto a través de las operaciones definidas al efecto por la máquina abstracta.
7. Un objeto deberá poder disminuir el número de permisos de una capacidad. Esta facilidad podrá aprovecharse para enviar capacidades restringidas a otros objetos, de los que sólo interesa que realicen un número de operaciones más reducido y de esta manera poder cumplir el principio de mínimo privilegio. Esto se consigue mediante la instrucción que permite restringir el acceso a un método en una capacidad (Restringir <capacidad>, <método>).
8. Cuando se crea un objeto, el objeto creador recibe una capacidad con permisos para todos sus métodos. Esto implica que estará activo el permiso de “salto de protección”.
9. Cuando un objeto restringe un permiso de una capacidad cuyo permiso de “salto de protección” está activo, entonces éste se desactiva y se crean permisos explícitos para todos los métodos, estando todos activos salvo el que se pretende restringir con

dicha operación. Posteriores restricciones de otros permisos implicarán únicamente la desactivación del permiso susodicho.

10. La fusión de capacidades y referencias implica la modificación interna de algunas operaciones del modelo de objetos relacionadas con las referencias. El uso de referencias debe tener en cuenta ahora que también existe la información de protección. Así por ejemplo, instrucciones como la asignación, deberán copiar también los permisos. La creación de un objeto debe retornar una capacidad con permisos de acceso inicial, etc. Sin embargo estas modificaciones internas son transparentes para las aplicaciones, puesto que su semántica externa no cambia.

14.3.5 Relación con la distribución

La integración de la información de protección en las referencias imposibilita su falsificación, dado que no hay forma de acceder a dicha información. En un entorno de una única máquina abstracta, ésta es la responsable de la comprobación de la protección en las operaciones de invocación a método.

Cuando se pasa a un entorno distribuido de máquinas abstractas, cabe preguntarse qué ocurre con la información de protección y con la comprobación de la misma. Más aún, será necesario garantizar que la información de protección pueda viajar de una máquina abstracta a otra de forma segura.

14.3.5.1 Estructura de las referencias en el sistema integral distribuido

La introducción de la distribución en el sistema integral no modifica en modo alguno la estructura de las referencias. La identificación del objeto, por un lado, permitirá conocer, a través de los objetos localizadores, la ubicación del objeto, allá donde quiera que esté. La información de protección se restringe a indicar qué métodos pueden ser ejecutados por el propietario de la referencia, información que sigue siendo válida con independencia de la ubicación de los objetos poseedor y referenciado.

14.3.5.2 Comprobación de la protección en el sistema integral distribuido

La comprobación de los permisos de invocación de los métodos es responsabilidad de las máquinas abstractas. Para invocar un método de un objeto, el objeto cliente usa la referencia (capacidad) de que dispone para el objeto, que incluye la información de protección, y proporciona también la identidad del método a invocar. En un sistema integral formado por una sola máquina abstracta, la comprobación de la protección se simplifica por el hecho de disponer localmente de toda la información necesaria. De una manera sencilla, la comprobación de la protección se limitaría a realizar una búsqueda del método invocado en la clase y las superclases del objeto referenciado, de tal forma que dicha búsqueda resulte en la identificación del permiso que hace referencia al método dentro de la capacidad. Si el permiso está activo, la invocación podrá tener lugar.

En un sistema integral distribuido es posible que el objeto cliente esté ubicado en un nodo, el objeto invocado en otro, y las clases y superclases de este último estén diseminadas por una serie de ellos.

La comprobación de permisos se realizará en el nodo en el que reside el objeto invocado, al que se ha hecho llegar una copia de la capacidad de que dispone el objeto cliente (ver capítulo 12). Realmente es el objeto trabajador, creado al efecto, el que obtiene la copia de la capacidad y transforma la invocación remota en una invocación local.

En el caso de que la capacidad tenga **activo el "salto de protección"**, no será necesario hacer comprobación de protección alguna. Como ya se ha comentado, este caso se producirá, en general, cuando un objeto invoque métodos de alguno de sus agregados, lo cual es bastante frecuente.

En el caso de que **se tenga que realizar la comprobación de los permisos** y dado que se trata de una operación de invocación, será necesario, en primer lugar, disponer de la definición del método a invocar. Como ya se mencionó en el capítulo 12, puede llegar a ser necesario crear localmente una copia del método y depositarla en la clase sombra. Si no es necesario crear una copia local será porque se dispone del original (la clase está disponible localmente) o ya existe una copia en la clase sombra. En cualquiera de los dos casos, la información necesaria para identificar el permiso en la capacidad está ya disponible. Si, por el contrario, es necesario obtener una copia del método y colocarla en la clase sombra, la misma operación de búsqueda del método en el sistema integral se encargará de retornar la información necesaria para la comprobación de los permisos.

Para cualquiera de las situaciones anteriores la comprobación de permisos tendrá lugar en el nodo en el que reside el objeto invocado. En el caso de que la invocación sea remota, será siempre necesario enviar el mensaje con los datos de la invocación antes de poder comprobar si ésta va a poder tener lugar, dado que en el nodo origen no es posible realizar, en principio, la comprobación de permisos.

14.3.5.3 Optimización de la comprobación de permisos

El coste introducido por aquellas invocaciones remotas para las cuales no se dispone de permiso puede verse reducido si en el nodo emisor se dispone de alguna información que permita conocer la clase del objeto invocado. En ese caso sería posible conocer si el objeto que realiza la invocación está autorizado a invocar el método. En caso de que no lo esté, el mensaje que contiene los datos de la invocación no se enviaría, y se podría retornar rápidamente una excepción.

Es necesario estudiar dónde almacenar la información de la clase del objeto invocado y quién va a ser el responsable de hacer la comprobación de permisos en el nodo origen.

En el nodo origen, la cache de direcciones de objetos no locales guarda información acerca de aquellos objetos que se usan con cierta frecuencia y no se encuentran ubicados en el nodo. Dado que los objetos que se almacenan en esa cache deben ser los más utilizados de manera remota, es un buen lugar para colocar los identificadores de las clases de los objetos remotos. Dichos identificadores se pueden obtener modificando el mensaje de devolución de resultados de las invocaciones a método, con el fin de que formen parte de los mismos.

El responsable de la comprobación de permisos en el nodo origen será el meta-objeto emisor. Dado que él es quien utiliza los servicios del objeto localizador y el que realmente construye y envía el mensaje de la invocación, lo lógico es que sea el responsable de tomar la decisión de si se envía o no dicho mensaje. Como ya se ha dicho, la decisión podrá tomarse si localmente se dispone de la información de la clase a la que pertenece el objeto invocado.

14.3.5.4 Transmisión de información de protección

La información de protección sólo tiene sentido dentro de las referencias (capacidades) y, por tanto, no existirá en ningún momento de manera aislada. La transmisión de la información de protección se producirá, entonces, cuando se

transmitan referencias, por lo que habrá que asegurar que dicha transmisión se realice de manera adecuada.

Las referencias viajan por la red por alguno de los motivos siguientes: en una invocación remota o en una operación de migración de un objeto. En cualquiera de los dos casos, los únicos objetos que tienen acceso real a la red son los objetos comunicador. Se proporciona en este punto un **primer nivel de seguridad**, dado que solamente aquellos objetos que tengan permisos para invocar a los objetos comunicador podrán acceder a la red. En general, solamente podrán hacerlo ciertos objetos del sistema operativo.

En segundo lugar, y como ya se comentó en el capítulo 12, existe la posibilidad de introducir un **proceso de cifrado** de los mensajes que se entregan a los objetos comunicador. Dicho cifrado introduciría un nivel superior de seguridad de la información que se transmite, y en particular, de las referencias que forman parte de esa información.

14.4 Resumen

Las características de persistencia y seguridad del modelo de objetos ya se establecieron en los primeros capítulos como muy importantes. Después de haber descrito los mecanismos del subsistema de distribución, es necesario ver de qué manera se relacionan con los propuestos para la persistencia y la seguridad.

La persistencia proporciona la abstracción de un espacio de objetos potencialmente infinito, en el que los objetos residen hasta que ya no son necesitados. Se puede considerar como una memoria virtual persistente para los objetos. Su combinación con la distribución permite extender dicho mundo de objetos, de manera que se puede ver como infinito en el espacio y en el tiempo. Un sistema operativo que proporciona los servicios de distribución y persistencia debe conseguir que toda referencia puede ser utilizada, de manera transparente, con independencia de la ubicación (local o remota) y situación (memoria volátil o persistente) del objeto que refiere.

Por su parte, todo sistema operativo moderno proporciona algún tipo de soporte para múltiples usuarios. En este caso, la protección de los recursos del sistema cobra especial relevancia. Dado que todos los recursos del sistema integral se ofrecen como objetos, la protección ha de ser implantada para garantizar su uso por los usuarios autorizados. La protección se lleva, entonces, al límite de identificar qué métodos de un objeto se pueden invocar en base a la capacidad (referencia) que al efecto se tiene que usar. Dado que las referencias son globales, no es necesario realizar ninguna modificación para la comprobación de permisos en las invocaciones remotas. De todas maneras, es necesario garantizar que la información de protección se transmite de manera segura a través de la red.

CAPÍTULO 15 EL SISTEMA INTEGRAL ORIENTADO A OBJETOS OVIEDO3

15.1 Introducción

Oviedo3 [CIA+96] es un proyecto investigación que está siendo desarrollado por un equipo de profesores y alumnos del grupo de investigación en Tecnologías Orientadas a Objetos de la Universidad de Oviedo, que pretende construir un sistema integral orientado a objetos con la arquitectura descrita en el capítulo 6.

Los elementos básicos que proporcionarán el espacio de objetos del sistema integral son la máquina abstracta Carbayonia y el sistema operativo SO4 [ATA+97] que extiende las capacidades de la misma.

La combinación de estos dos elementos forma el sistema integral orientado a objetos (SIOO) Oviedo3. Esta plataforma permite dar un soporte directo a las tecnologías orientadas a objetos, y desarrollar más rápidamente todos los demás elementos de un sistema de computación. Todos estos elementos, desde la máquina hasta la interfaz de usuario utilizarán el mismo paradigma de la orientación a objetos.

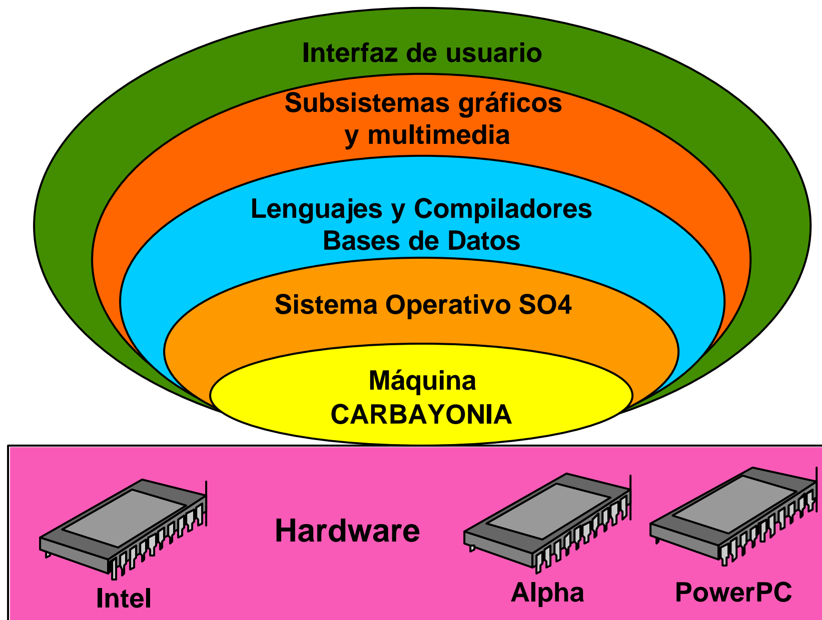


Figura 15.1. Esquema general del sistema integral orientado a objetos Oviedo3

Los fines del proyecto Oviedo3 son tanto didácticos como de investigación. En lo que a investigación se refiere, se pretende usar el sistema como plataforma de investigación para las diferentes áreas de las tecnologías de objetos en un sistema de computación: la propia máquina abstracta y sistema operativo orientado a objetos, así

como lenguajes y compiladores OO, bases de datos OO, sistemas gráficos y multimedia, interfaces de usuario, etc.

15.2 La máquina abstracta carbayonia

Carbayonia es una máquina abstracta orientada a objetos que sigue la estructura de referencia marcada en el capítulo 7. La descripción de la máquina está adaptada de la documentación del primer prototipo de la misma, desarrollado como proyecto fin de carrera de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Oviedo [Izq96].

La máquina es una máquina abstracta orientada a objetos pura que implementa el modelo único de objetos del sistema. Todos los objetos tienen un identificador que se usa para operar con ellos exclusivamente a través de referencias. Las referencias, como los propios objetos, tienen un tipo asociado y tienen que crearse y destruirse. En Carbayonia todo se hace a través de referencias: las instrucciones tienen referencias como operandos; los métodos de las clases tienen referencias como parámetros y el valor de retorno es una referencia.

15.2.1 Estructura

Para comprender la máquina Carbayonia se utilizan las tres áreas de la máquina de referencia. Un área se podría considerar como una zona o bloque de memoria de un microprocesador tradicional. Pero en Carbayonia no se trabaja nunca con direcciones físicas de memoria, si no que cada área se puede considerar a su vez como un objeto el cual se encarga de la gestión de sus datos, y al que se le envía mensajes para que los cree o los libere.

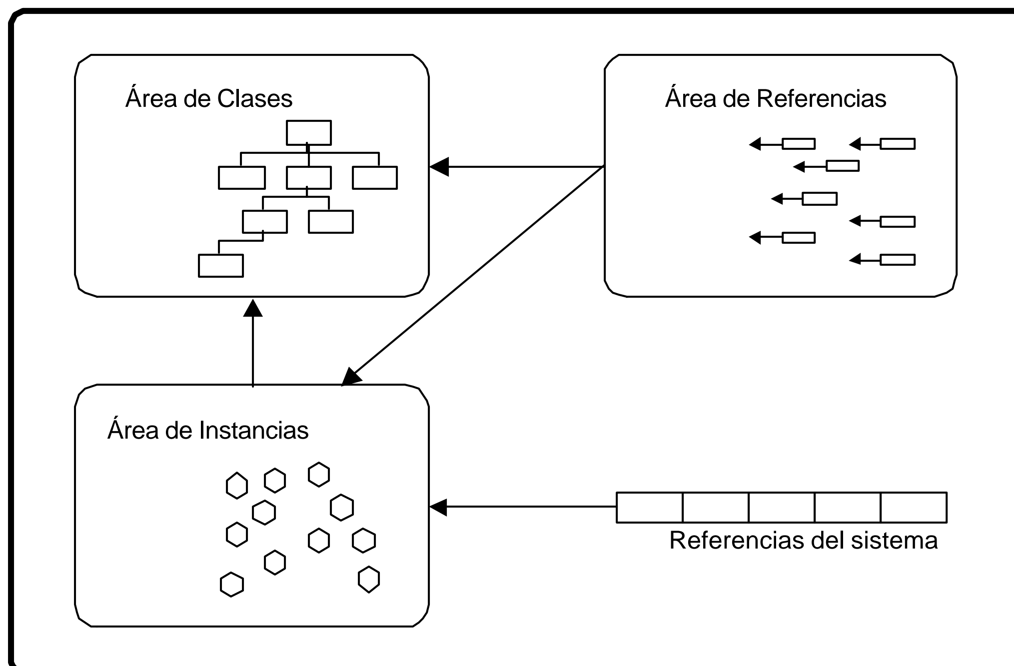


Figura 15.2. Estructura de referencia de una máquina abstracta para el sistema integral.

A continuación se expone una breve descripción de las áreas que componen Carbayonia, comparándola para una mejor comprensión con arquitecturas convencionales como las de Intel x86. Sus funciones se verán mas en detalle en la descripción de las instrucciones.

15.2.1.1 Área de Clases

En éste área se guarda la descripción de cada clase. Esta información está compuesta por los métodos que tiene, qué variables miembro la componen, de quién deriva, etc. Esta información es fundamental para conseguir propiedades como la comprobación de tipos en tiempo de ejecución (RTTI, *Run Time Type Information*), la invocación de métodos con polimorfismo, etc.

Aquí ya puede observarse una primera diferencia con los micros tradicionales, y es que en Carbayonia realmente se guarda la descripción de los datos. Por ejemplo, en un programa en ensamblador del 80x86 hay una clara separación entre instrucciones y directivas de declaración de datos: las primeras serán ejecutadas por el micro mientras que las segundas no. En cambio Carbayonia ejecuta (por así decirlo) las declaraciones de las clases y va guardando esa descripción en éste área.

15.2.1.2 Área de Instancias

Aquí es donde realmente se almacenan los objetos (instancias de las clases). Cuando se crea un objeto se deposita en éste área, y cuando éste se destruye se elimina de aquí. Se relaciona con el área de clases puesto que cada objeto es instancia de una clase determinada. Así desde un objeto se puede acceder a la información de la clase a la que pertenece.

Las instancias son identificadas de forma única con un número que asignará la máquina en su creación. La forma única por la que se puede acceder a una instancia es mediante una referencia que posee como identificador el mismo que la instancia. Se mantiene el principio de encapsulamiento. La única forma de acceder a una instancia mediante una referencia es invocando los métodos de la instancia.

15.2.1.3 Área de Referencias

Para operar sobre un objeto necesitamos antes una referencia al mismo. En éste área es donde se almacenan dichas referencias. El área de referencias se relaciona con el área de clases (ya que cada referencia tiene un tipo o clase asociado) y con el área de instancias (ya que apuntan a un objeto de la misma). Una referencia se dirá que está *libre* si no apunta a ningún objeto. Las referencias son la única manera de trabajar con los objetos[17].

15.2.1.4 Referencias del Sistema

Son una serie de referencias que están de manera permanente en Carbayonia y que tienen funciones específicas dentro del sistema. En éste momento simplemente se dará una breve descripción, ya que cada una se explicará en el apartado apropiado.

this: Apunta al objeto con el que se invocó el método en ejecución.

exc: Apunta al objeto que se lanza en una excepción.

rr (*return reference*): Referencia donde los métodos dejan el valor de retorno.

---

[17] Por tanto, aunque en algunos casos se mencionen los objetos directamente, como por ejemplo "se devuelve un objeto de tipo cadena" se entiende siempre que es una referencia al objeto.

15.2.2 El lenguaje Carbayón: juego de instrucciones

El juego de instrucciones de la máquina se describe en términos del lenguaje ensamblador asociado al mismo. Este lenguaje se denomina Carbayón y será la interfaz de las aplicaciones con la máquina. En cualquier caso, existe la posibilidad de definir una representación compacta de bajo nivel (*bytecode*) de este lenguaje que sea la que realmente se entregue a la máquina. Una descripción completa del lenguaje Carbayón se puede encontrar en el Anexo A.

A continuación se muestra un **ejemplo de programación** en lenguaje Carbayón. Se trata de la definición de una clase `ServidorNombres` que proporciona cierta funcionalidad de un servidor de nombres. Dispone de dos métodos: `setTamanio`, para inicializar el servidor y `registrarRef`, para registrar un nombre simbólico a asociar a una referencia a objeto.

```
CLASS ServidorNombres
aggregation
      nelems:integer;
      tabla:array;
Methods
setTamanio()
instances
      cero:integer(0);
code
      tabla.setsize(cero);
      nelems.set(cero);
      exit;
endcode

registrarRef(nom:string;ref:object)
refs
      tam:integer;
instances
      uno:integer(1);
      dos:integer(2);
code
      tabla.getsize():tam;
      tam.add(dos);
      tabla.setsize(tam);
      tabla.setref(nelems,nom);
      nelems.add(uno);
      tabla.setref(nelems,ref);
      nelems.add(uno);
      exit;
endcode
ENDCLASS
```

15.2.3 Características de las clases Carbayonia

A continuación se describe brevemente las características propias de las clases Carbayonia.

15.2.3.1 Herencia virtual

Un aspecto a destacar es que toda derivación es virtual. Al contrario que en C++, no se copian simplemente en la clase derivada los datos de las superclases. Al retener toda la semántica del modelo de objetos en tiempo de ejecución, simplemente se mantiene la información de la herencia entre clases. Es decir, en la estructura de herencias tipo como la de la figura, la clase D sólo tiene una instancia de la clase A. Se mantiene sincronizada respecto a los cambios que se puedan hacer desde B y desde C. A la hora de crear instancias de una clase, se repite la misma estructura.

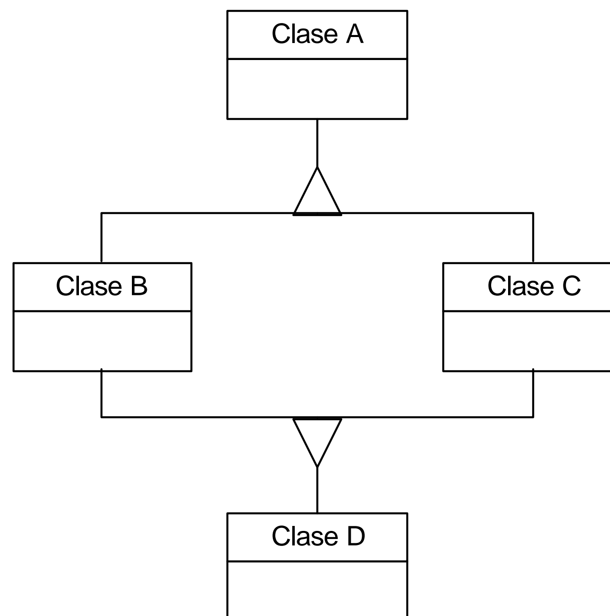


Figura 15.3. Jerarquía de herencia múltiple con ancestro compartido

15.2.3.2 Herencia múltiple. Calificación de métodos

El problema de la duplicidad de métodos y variables en la herencia múltiple se soluciona dando prioridad a la superclase que aparece antes en la declaración de la clase. Si en la figura anterior se supone que tanto la clase B como la clase C tienen un método llamado M, y desde D (o a través de una referencia a a D) se invoca a dicho método, se ejecutará el método de la clase que primero aparezca en la sección Isa de la clase D. En caso de que se desee acceder al otro método se deberá calificar el método, especificando antes del nombre del método la clase a la que pertenece, separada por dos puntos.

a.Metodo()	// método de la clase B
a.ClaseB:Metodo()	// método de la clase B
a.ClaseC:Metodo()	// método de la clase C

15.2.3.3 Uso exclusivo de métodos

No existen operadores (véase la clase básica Integer) con notación infija. Los operadores son construcciones de alto nivel que se implementarán como métodos. Esto es lo que hace al fin y al cabo el C++ a la hora de sobrecargarlos.

15.2.3.4 Uso exclusivo de enlace dinámico (sólo métodos virtuales)

No es necesario especificar si un método es virtual[18] o no, ya que todos los métodos son virtuales (polimórficos). Es decir, se utiliza únicamente el mecanismo de enlace dinámico, siguiendo la línea de una arquitectura OO más pura. El uso de enlace estático restringe en exceso la extensibilidad del código, perdiéndose una de las ventajas de la OO. La posible sobrecarga de ejecución de los métodos virtuales se puede compensar con una implementación interna eficiente.

15.2.3.5 Ámbito único de los métodos

No se restringe el acceso a los métodos clasificándolos en ámbitos como los private, public o protected del C++. Estos accesos son de utilidad para los lenguajes de alto nivel pero para una máquina no tienen tanto sentido. Si un compilador de alto nivel no desea que se accedan a unos determinados métodos private, lo único que tiene que hacer es no generar código de llamada a dichos métodos. Un ejemplo parecido ocurre cuando se declara una variable const en C++. No es que la máquina subyacente sepa que no se puede modificar; es el compilador el que no permite sentencias que puedan modificarla.

En cualquier caso, en el sistema operativo existe un mecanismo de protección (véase el capítulo 14) que permitirá una restricción de acceso individualizada para cada método y cada objeto. Así se podrán crear ámbitos de protección particularizados para cada caso, en lugar de simplemente en grupos private, public y protected.

15.2.3.6 Inexistencia de constructores y destructores

No existen métodos especiales caracterizados como constructores ni destructores. Al igual que ocurre con algunos de los puntos anteriores, es el lenguaje de alto nivel el que, si así lo desea, debe generar llamadas a unos métodos que hagan dichas labores a continuación de las instrucciones de Carbayonia de creación y destrucción de objetos.

Sin embargo, no es necesario que se aporten métodos para la gestión de la semántica de los objetos agregados[19], que ya es conocida por la máquina y se realiza automáticamente.

15.2.3.7 Redefinición de métodos

Para que un método redefina (*overriding*) a otro de una superclase debe coincidir exactamente en número y tipo de parámetros y en el tipo del valor de retorno. No se permite la sobrecarga (*overloading*) de métodos (dos o más métodos con el mismo nombre y diferentes parámetros).

15.2.4 Jerarquía de clases básicas

Independientemente de las clases que defina el programador (y que se irán registrando en el área de clases), Carbayonia tiene una serie de clases básicas (también denominadas primitivas) que se pueden considerar como definidas en dicho área de manera permanente.

---

[18] En el sentido de C++ de la palabra virtual: utilizar enlace dinámico con ese nombre de método.

[19] Como por ejemplo su eliminación al eliminarse el objeto que los contiene.

- Object
- Bool
- Integer
- Float
- String
- Array

Estas clases básicas serán las clases fundamentales del modelo único que se utilizarán para crear el resto de las clases. Una aplicación Carbayonia es un conjunto de clases con sus métodos en los cuales se llaman a otros métodos. Siguiendo este proceso de descomposición, siempre llegamos a las clases básicas y a sus métodos.

En cualquier caso, estas clases básicas no se diferencian en nada de cualquier otra clase que cree el usuario. Desde el punto de vista de utilización son clases normales como otras cualesquiera. Cada implementación de la máquina establecerá los mecanismos necesarios para proporcionar la existencia de estas clases básicas.

Las clases básicas se organizan en una jerarquía, cuya raíz es la clase básica Object.

En el Apéndice B se describe de manera exhaustiva el conjunto de clases anterior.

15.2.5 Clase básica objeto

Se define clase básica como aquella cuya implantación está codificada internamente en la propia máquina abstracta, utilizando para ello el lenguaje de desarrollo empleado para codificar la máquina.

Dentro de la jerarquía de clases básicas ofertadas por la máquina abstracta, se encuentra la clase **Object**, de la que derivan todos los objetos existentes en el sistema, bien sean de usuario o de otras capas superiores como el sistema operativo, bases de datos, etc.

15.2.5.1 Nivel de abstracción único: Uniformidad en torno al objeto

Existen dos características de la máquina abstracta que garantizan la uniformidad en torno a la abstracción de objeto, o lo que es lo mismo, impiden que la máquina proporcione ninguna otra abstracción en tiempo de ejecución, excepto el objeto.

Juego de instrucciones

Por un lado, el juego de instrucciones reducido que la máquina abstracta ofrece pocas posibilidades al usuario para añadir elementos nuevos en tiempo de ejecución.

Concretamente, las únicas instrucciones que la máquina ofrece para añadir elementos son la **instrucción declarativa** Class para añadir una clase o bien, la **instrucción** new para crear una instancia de una clase, o sea, un objeto.

Por tanto, las operaciones que la interfaz de la máquina abstracta permite realizar en tiempo de ejecución, se limitan a aquellas referidas a la definición, creación y manipulación de objetos.

Es decir, en tiempo de ejecución, el único elemento que existe es el objeto y, para resolver un problema del mundo real, la máquina abstracta permite crear objetos e invocar métodos en los mismos. No existe ninguna otra abstracción en tiempo de ejecución.

Clase básica

Por otro lado, la máquina abstracta define un objeto como una instancia en tiempo de ejecución de una clase, creada gracias a la instrucción new[20].

Toda clase pertenece a la jerarquía de clases de la máquina, que tiene como raíz la clase Object. Esta clase define el comportamiento básico de cualquier nuevo elemento que se cree en el sistema.

La definición de una clase básica de objetos de la que derivan todos los objetos existentes en el sistema, promueve que todos los objetos estén dotados de una serie de características básicas que, ineludiblemente, heredarán.

15.2.5.2 Identificador único de objetos

Todo objeto que existe en el sistema en tiempo de ejecución es una instancia de clase básica Object y se comportará según el comportamiento definido para él por la máquina abstracta, al tratarse esta de una clase básica.

Cuando se invoca la instrucción new sobre cualquier clase de la jerarquía de clases de la máquina (sean estas clases básicas o de usuario), la máquina abstracta crea una nueva instancia de un objeto al que proporciona un identificador único y global en el sistema [ATD+98a, ATD+98b].

15.2.5.3 Reflectividad: Aumento del soporte al objeto por parte de la máquina abstracta

El aumento en la semántica del objeto supone un soporte más sofisticado al objeto. Dicho de otra forma, la representación del objeto que la máquina soporta se ve modificada por la necesidad de mantener la relación entre los objetos y el conjunto de meta-objetos que completan su entorno de ejecución.

Soporte en el entorno de ejecución base al enlace base-meta

Se añade a la clase básica Object, la referencia **meta**. De esta forma, las instancias en tiempo de ejecución podrían acceder a su meta-espacio simplemente consultando a qué instancia apunta la referencia meta.

Así, pueden determinar a qué objetos pasarle el control cuando la máquina tenga lagunas en el proceso de ejecución. También se hace posible cambiar el meta-objeto simplemente cambiando el meta-objeto al que apunta la referencia meta.

Realmente, este enlace meta no será una única entrada sino un conjunto de entradas a los distintos meta-objetos que definen el entorno en tiempo de ejecución del objeto base, a saber, los meta-objetos encargados del paso de mensajes, los meta-objetos encargados del control de la concurrencia y aquel o aquellos encargados de la planificación.

Este punto de entrada al meta-espacio se puede representar a su vez como un objeto, al estilo del reflector de Apertos [Yok92] o Merlin [AK95].

---

[20] Se puede considerar que la instrucción new es equivalente al envío del mensaje new al objeto clase concreto del que se pretende crear una instancia.

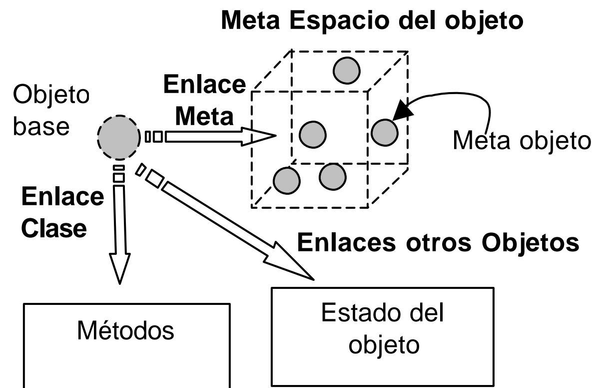


Figura 15.4. Enlace meta: asociación del objeto y su meta espacio

15.3 El sistema operativo SO4

El sistema operativo SO4 [ATA+96, ATA+97] es el encargado de extender la funcionalidad básica de la máquina abstracta Carbayonia. Los elementos más importantes de esta funcionalidad son los que permiten dotar a los objetos de manera transparente con las propiedades de persistencia, concurrencia, distribución y seguridad.

La persistencia permite que los objetos permanezcan en el sistema hasta que ya no sean necesitados. La distribución hace que se pueda invocar un método de un objeto independientemente de la localización del mismo en el sistema integral. La concurrencia proporciona al modelo de objetos la capacidad de ejecutar tareas en paralelo. Otro elemento muy importante que debe estar presente en un sistema distribuido con múltiples usuarios es la seguridad o protección del acceso a los objetos. Sólo los objetos autorizados podrán enviar mensajes a otros objetos.

El objetivo fundamental a la hora de implantar estas propiedades es la transparencia. Los objetos adquirirán estas propiedades sin hacer nada especial. No tienen por qué ser conscientes ni intervenir en los mecanismos que se usan para lograrlas (excepto en el caso en que sea imprescindible, como los objetos del sistema operativo). Por otro lado es muy importante que la introducción de estas propiedades se integre de manera fluida con el modelo de objetos del sistema. Es decir, que no se necesiten cambios en la semántica del modelo, o que sean menores y no rompan la esencia del modelo.

Para alcanzar estos objetivos, el sistema operativo utilizará el mecanismo de reflectividad proporcionado por la máquina abstracta. La arquitectura del sistema, basado en la máquina abstracta, ciertas propiedades del modelo de objetos, la reflectividad y la extensión en el espacio del usuario facilitan la implementación del sistema operativo, para alcanzar el objetivo de un mundo de objetos: un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados.

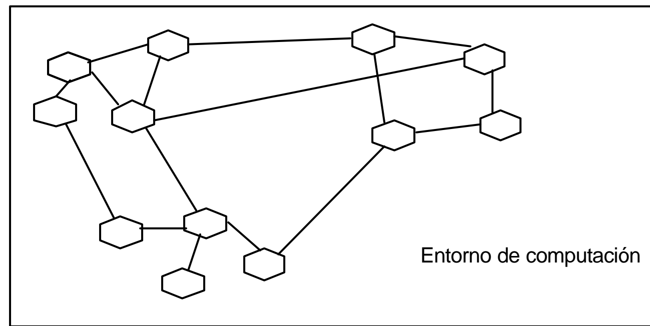


Figura 15.5. Entorno de computación compuesto por un conjunto de objetos homogéneos.

El diseño en profundidad de las propiedades que debe implementar el sistema operativo, así como la implementación de las mismas está siendo investigado por otros miembros del proyecto Oviedo3. Los resultados de dichas investigaciones se pueden encontrar en [Álv98, Día2000, Taj2000].

15.4 Resumen

Oviedo3 es el nombre del Sistema Integral Orientado a Objetos objeto del proyecto de investigación del mismo nombre que se está realizando en el seno del grupo de investigación de Tecnologías Orientadas a Objetos de la Universidad de Oviedo.

El espacio de objetos del sistema integral queda definido por su máquina abstracta, denominada Carbayonia, y su sistema operativo, denominado SO4. Dentro de SO4 se enmarca la distribución de objetos, aspecto central de esta Tesis.

La máquina Carbayonia, de la que existen varias versiones. sigue la arquitectura de referencia descrita en el capítulo 7. Entre otras cosas, su conjunto de instrucciones describe una lenguaje ensamblador, de alto nivel y orientado a objetos, denominado Carbayón. La reflectividad de la máquina será la herramienta fundamental para su colaboración con el sistema operativo.

El sistema operativo SO4 extiende la funcionalidad básica de la máquina abstracta Carbayonia. Los elementos más importantes de esta funcionalidad son los que permiten dotar a los objetos de manera transparente con las propiedades de persistencia, concurrencia, seguridad y, evidentemente, distribución.

CAPÍTULO 16 IMPLEMENTACIÓN DE UN PROTOTIPO DE DISTRIBUCIÓN DE OBJETOS SOBRE EL SISTEMA INTEGRAL OVIEDO3

16.1 Introducción

Para construir el sistema de distribución de objetos del sistema integral se partió de una versión centralizada de la misma [Izq96] y que no presentaba la característica de reflectividad requerida. Se implementó una versión preliminar del sistema de distribución de objetos AGRA con el fin de estudiar los problemas preliminares que presentaba la introducción de dicho mecanismo en un entorno de máquinas abstractas OO. Aunque los resultados fueron satisfactorios, no se cumplieron todas las expectativas, dado que los mecanismos y políticas de distribución estaban fijados fuertemente en la implementación del sistema.

Ha sido necesario, por tanto, modificar la implementación de la máquina abstracta para dotarla de una arquitectura reflectiva y así, construir el sistema de distribución diseñado en esta tesis.

Se describen en este capítulo las dos versiones realizadas del sistema de distribución, dado que la primera ayudará de manera importante a comprender las dificultades encontradas para introducir un mecanismo tan complejo como la distribución de objetos en un entorno de máquinas abstractas que tenían establecido una interfaz (lenguaje Carbayón) que no podía ser modificado. La segunda versión complementa a la primera en el sentido de que permite ver cómo es posible integrar de manera elegante y transparente la distribución en el sistema integral tal y como aparece descrito en esta tesis.

16.2 Presentación de la máquina abstracta Carbayonia

En este apartado se describe el prototipo de la máquina abstracta Carbayonia (simulador por software de la máquina) desarrollado como proyecto fin de carrera de la Escuela Superior de Ingenieros Informáticos de la Universidad de Oviedo. Se puede consultar los detalles de la implantación en [Les00].

16.2.1 Diseño de la máquina como conjunto de objetos

La aplicación intensiva de los principios de diseño expuestos con anterioridad, da como resultado un prototipo en el que, todos y cada uno de los elementos que constituyen la máquina, desde los elementos arquitectónicos, como pueden ser las áreas de clases o instancias, hasta las instrucciones que definen el ensamblador de la máquina, se representan en el simulador mediante objetos (TClassArea, TInstanceArea, TInstruction, etc.). Cada uno de estos objetos definen una parte del comportamiento de la máquina mediante un conjunto de métodos.

16.2.1.1 Diagrama de clases general

El resultado de aplicar el principio de OO en la propia construcción de la máquina, es una estructura en tiempo de ejecución de la misma en la que se representa, por medio de objetos en tiempo de ejecución, la estructura del programa en Carbayonia. Esto se muestra en el diagrama general de clases del prototipo que se puede ver en la figura 16.1.

Figura 16.1. Diagrama de clases del intérprete.

16.3 Descripción de la jerarquía de clases

A la hora de describir la implantación del SIOO es necesario describir la **jerarquía de clases del prototipo** y las relaciones entre ellas.

En el diseño e implantación de la máquina abstracta Carbayonia se aplicaron a fondo todos los principios de la OO siendo la herencia y el polimorfismo claves en el diseño. Esto facilita considerablemente la definición de la jerarquía de clases ya que, por un lado, la herencia permite definir un objeto abstracto, raíz de la jerarquía, a partir del cual, mediante refinamientos sucesivos, se pueden derivar distintas especializaciones. Por su parte, el polimorfismo, permite que el hecho de qué clase se trate sea totalmente transparente al resto del sistema.

Existe, por tanto, una jerarquía de clases para cada uno de los elementos arquitectónicos de la máquina: las clases, las instancias, los métodos, las instrucciones, etc. En [Les00] se muestra la jerarquía de clases mencionada de manera completa. A continuación, y a modo de muestra, se presenta la relativa al elemento arquitectónico clase.

El objeto abstracto raíz de la jerarquía, TClass, representa cualquier tipo de clase, y de él heredarán otros que representen clases primitivas (un TCxxx por cada clase primitiva) y de usuario. La lista de las clases primitivas se muestra en el capítulo 15 y, de una manera más exhaustiva, en [Izq96].

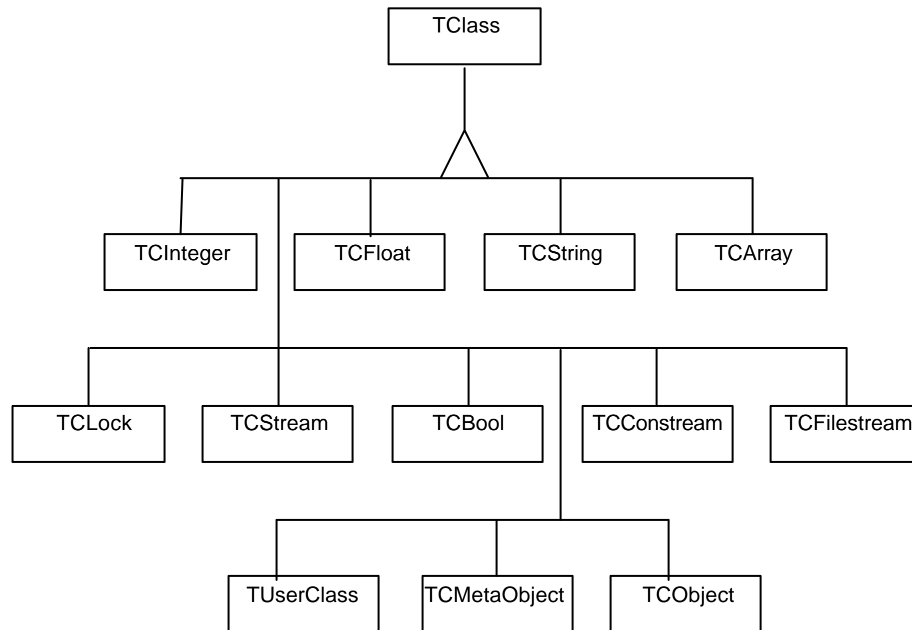


Figura 16.2. Jerarquía de TClass para las clases con implantación primitiva.

16.4 Prototipo de distribución no reflectivo

El primer prototipo de AGRA se desarrolló como Proyecto Fin de Carrera de la Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de la Universidad de Oviedo sobre el Sistema Operativo Windows NT. La descripción completa del mismo se puede consultar en [Fer98].

Este prototipo se basó en la versión de la máquina abstracta que aún no disponía de la característica de reflectividad [Izq96], por lo que los mecanismos y políticas implementados no podían ser modificados sino era a través de la modificación de la propia máquina abstracta. De todas formas, sirvió para ir detectando problemas que presentaba el diseño interno de la máquina y su modificación para la segunda versión del prototipo de distribución.

El objetivo principal que se seguía en esta primera versión era dotar de características de distribución a un sistema integral formado por varias máquinas abstractas de manera transparente a los objetos de usuario, es decir, la interfaz proporcionada a los objetos de usuario a través del juego de instrucciones de la máquina no podía ser modificado, permitiendo el funcionamiento de los programas tanto en una máquina abstracta de la primera generación (centralizada) como en un entorno de máquinas abstractas como las que se estaban construyendo. Esta primera versión se construyó permitiendo la invocación remota de métodos y la propagación distribuida de excepciones, aunque sacrificando la migración de objetos y la posibilidad de modificar y adaptar los mecanismos.

16.4.1 Problemas presentados para la distribución por la máquina abstracta original

El primer problema que presentaba la máquina original para permitir la introducción de la distribución era la **utilización intensiva de punteros** que sustituían internamente a las referencias a objetos. En el caso de una única máquina abstracta, esta opción es muy

eficiente, dado que todos los objetos residen en un único espacio de direcciones. Al pasar a un entorno de varias máquinas abstractas, la solución de los punteros deja de ser válida. Un objeto que se desea utilizar en una máquina abstracta puede residir en cualquiera de las máquinas del sistema integral. Es necesario, por tanto, modificar el diseño interno de la máquina abstracta para que los objetos sean referenciados por su identificador de objeto y, a través de dicho identificador, se acceda finalmente a él.

El segundo problema para la distribución es el **modelo de objetos pasivo** utilizado en la primera versión de la máquina. Los métodos se ejecutan sobre hilos (*threads*), formados básicamente por una pila de contextos. Cada contexto de la pila almacena un puntero al método que lo creó así como sus variables locales. Un conjunto de llamadas a métodos anidadas dará lugar al apilamiento sucesivo de capas de contexto en la pila del hilo.

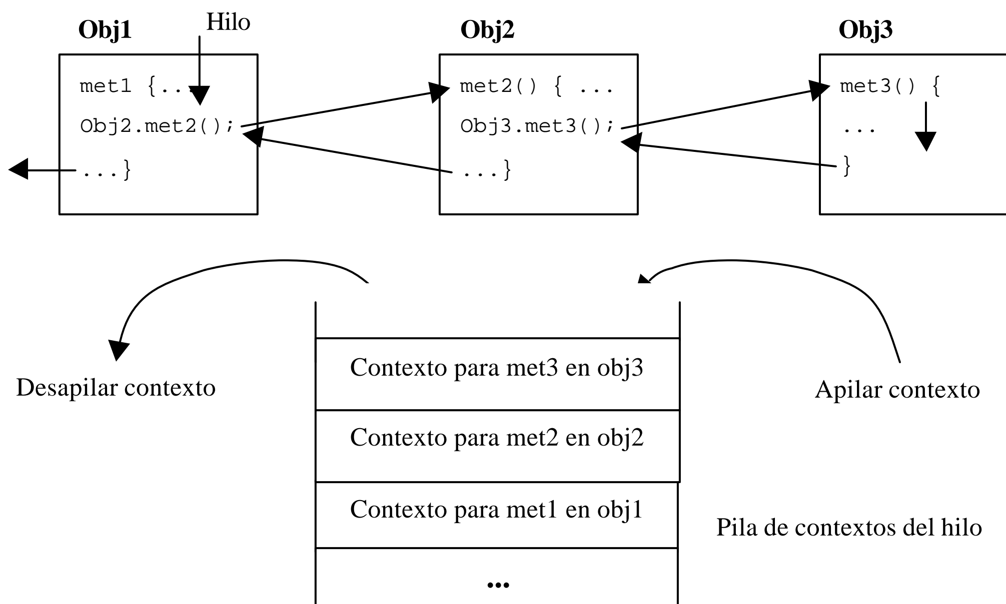


Figura 16.3. Relación entre métodos, contextos e hilo.

Para conseguir la invocación remota, basta con posibilitar que distintos contextos del mismo hilo puedan residir en máquinas abstractas distintas, de tal forma que cuando se realiza una invocación remota, el contexto creado se apila en una pila auxiliar de dicha máquina remota. De la misma manera, cuando se desapila un contexto, si se trata del último contexto de la pila, es necesario comprobar si dicha pila se creó debido a una invocación remota. Si es así, el hilo continuará ejecutándose sobre el contexto en que se realizó dicha invocación, ubicado en otra máquina abstracta.

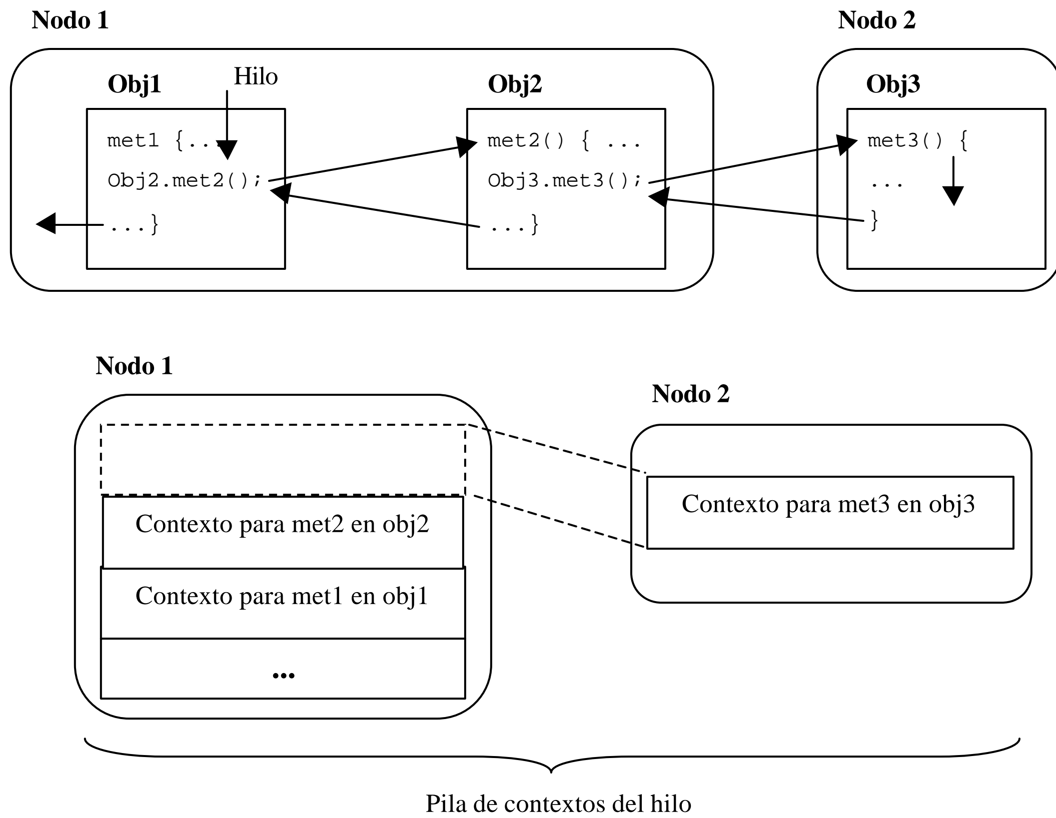


Figura 16.4. Relación entre métodos, contextos e hilo en una invocación remota.

La migración de objetos, tal como se ha planteado en esta tesis, no se puede conseguir de una manera sencilla con un modelo de objetos pasivo. Dado que se desea mover estado y computación de un objeto cuando se migra, la posible dispersión de la computación de un objeto en múltiples pilas de diferentes hilos dificulta notablemente la migración. La figura siguiente muestra un ejemplo de tal dispersión, donde dos hilos distintos incluyen como parte de su computación la invocación de un método del objeto `objx`. Si se desease migrar dicho objeto, sería necesario extraer los contextos relativos a su computación de las pilas correspondientes.

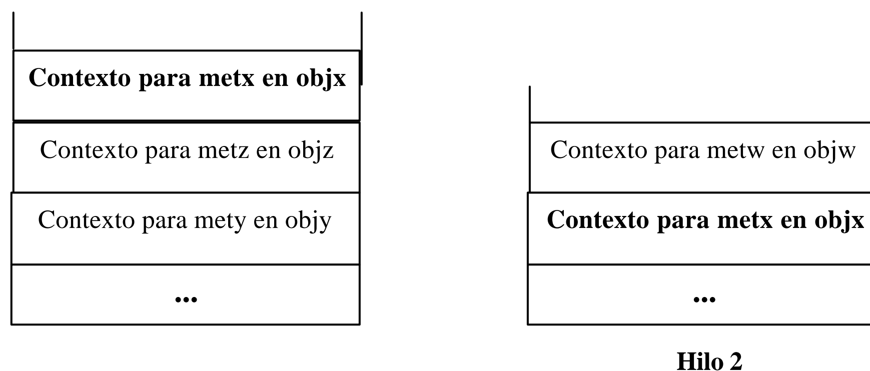


Figura 16.5. Dispersión de la computación de un objeto en múltiples hilos.

En definitiva, las dificultades encontradas en la primera versión de la máquina abstracta para introducir la distribución de una manera no traumática fueron

importantes. De ahí, que se optara por introducir únicamente la invocación remota de métodos como característica de la distribución.

16.4.2 Aspectos iniciales de diseño

Como se ha comentado, la utilización intensiva de punteros en la implementación de la máquina abstracta imposibilitaba la introducción de la distribución. La decisión a tomar en este sentido fue sustituir los punteros por identificadores para todos aquellos elementos susceptibles de residir en cualquier máquina abstracta y ser referenciados desde cualquier ubicación: objetos, clases y métodos tendrán un identificador que los distinguirá de manera unívoca en todo el sistema integral.

Una consecuencia inmediata de la sustitución de punteros por identificadores es la necesidad de introducir mecanismos de búsqueda de los elementos ahora referenciados por su identificador: la búsqueda se realizará, en primer lugar, en la máquina abstracta local y, si no se encuentra el elemento buscado, la búsqueda continuará por el resto de las máquinas abstractas.

La comunicación entre las distintas máquinas abstractas se realizó con llamadas a procedimientos remotos (RPC, *Remote Procedure Call*), dado que proporciona una interfaz más potente que la proporcionada por los *sockets*, de un nivel de abstracción demasiado bajo.

La nueva implementación de la máquina abstracta debe ser capaz de ejecutar métodos de objetos a la vez que escucha a través de la red las posibles invocaciones a método entrantes. Para conseguir ambas cosas, se dotó a la máquina abstracta de una arquitectura que constaba de varios hilos de Windows NT.

16.4.3 Invocación remota

La invocación a método es una instrucción de la máquina abstracta, con la siguiente sintaxis:

```
<ref>.<método>([params])
```

La invocación remota tiene lugar, de manera transparente, cuando el objeto invocado (el apuntado por *ref*) no se encuentra ubicado en la misma máquina abstracta que el objeto que le invoca.

La figura siguiente muestra el proceso general de invocación de un método. En el caso de que la invocación sea local, todas las operaciones mostradas se realizan dentro de la misma máquina abstracta.

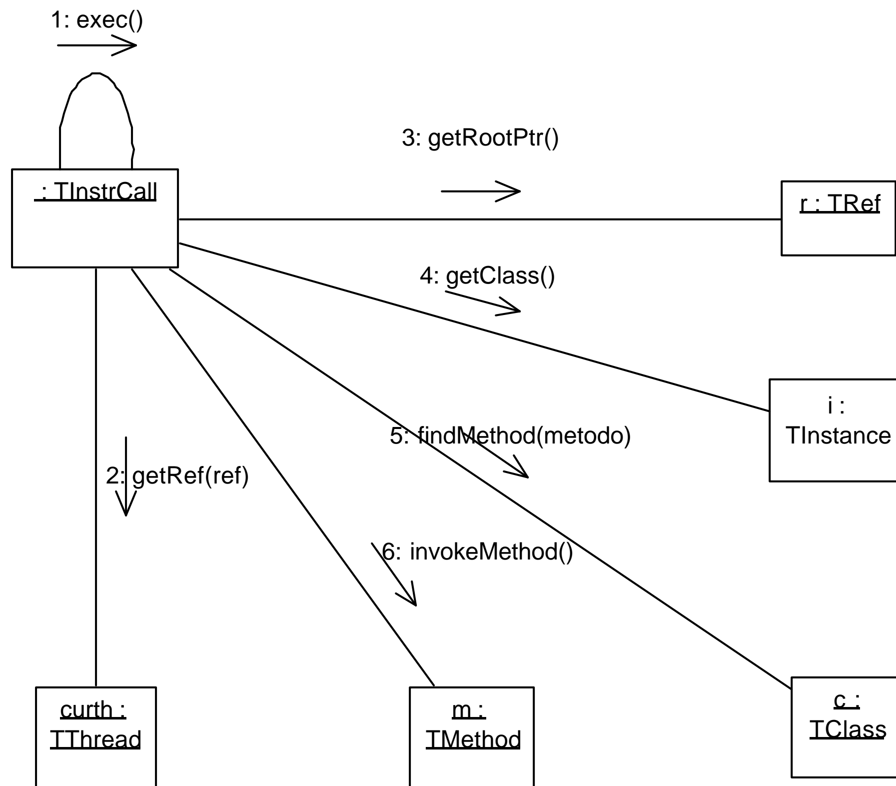


Figura 16.6. Ejecución de la instrucción Call

En primer lugar se busca la referencia a la que quiere invocar el método (2), se obtiene la instancia real subyacente (3) y se obtiene su clase (4) para así poder conocer el método que se desea invocar. A continuación se invoca a FindMethod de la clase (5) que devuelve el TMethod correspondiente, el cual ya se puede invocar (6).

En una invocación remota, el paso (2) se realiza siempre de manera local. La primera parte del paso (3) se corresponde con la búsqueda de la instancia en el área de instancias local. Si existe, se procede como se ha mencionado. Si no existe localmente, se construye un mensaje que contiene la información de la invocación y se envía a la máquina en la que está ubicada la instancia. En dicha máquina tienen lugar los pasos (4), (5) y (6) mostrados.

16.4.4 Propagación de excepciones

El tratamiento de excepciones estaba incluido en la primera versión de la máquina abstracta. Al pasar a una arquitectura distribuida, es necesario seguir soportando las excepciones de una manera transparente para los programas de usuario, de tal forma que las excepciones puedan seguir produciéndose, propagándose y manejándose de una manera similar al caso inicial.

Las excepciones son generadas por la máquina abstracta o por el usuario. Las generadas por la máquina abstracta responden a una utilización inadecuada de las instrucciones o los operandos que estas manejan. Por ejemplo, la instrucción *delete* genera una excepción si se le pasa una referencia a todo aquello que no sea un objeto válido (NULL, objeto que es agregado de otro, etc.). Las excepciones de usuario son generadas explícitamente por éste cuando se ejecuta la instrucción *throw*. En cualquier caso, el usuario puede haber definido un manejador de excepciones (instrucción

handler) que se encargará de redirigir el flujo de ejecución hacia una determinada parte del código en el caso de que se produzca una excepción.

Toda excepción se propaga desde el método en el que se produce y hacia atrás en la cadena de invocaciones a métodos hasta que se encuentra un manejador de interrupciones, desapilando los contextos que sean necesarios. En un entorno distribuido, dicha propagación puede tener que superar fronteras entre diferentes máquinas abstractas.

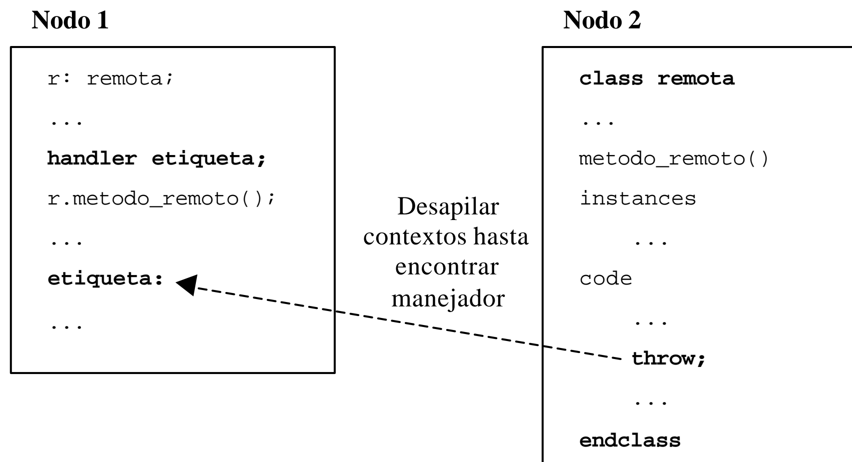


Figura 16.7. Funcionamiento de las excepciones.

Dado que los contextos pertenecientes al mismo hilo conocen cual es el contexto que les precede en la pila (aunque estén en máquinas distintas), la propagación de las excepciones únicamente tiene que seguir dicha pila "distribuida". Cuando se produce la excepción en el nodo remoto, se devuelve inmediatamente el control al nodo local con un mensaje, que contiene el código y el tipo de la excepción, para que continúe la búsqueda del manejador.

16.5 Prototipo de distribución reflectivo

El prototipo reflectivo de AGRA se desarrolla en dos Proyectos Fin de Carrera de la Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de la Universidad de Oviedo. La descripción completa de los mismos se puede consultar en [Les00, Rod00].

El prototipo descrito en [Rod00] se basó en la versión de la máquina abstracta que aún no disponía de la característica de reflectividad [Izq96], y su principal objetivo fue dotarla de dicha característica, permitiendo de esta forma la intervención del usuario en algunos aspectos del funcionamiento de la máquina.

El segundo prototipo, descrito en [Les00], basándose en el anterior, se encarga de dotar al sistema integral de la característica de distribución, siguiendo las líneas marcadas en esta Tesis.

En este apartado se muestran los resultados de ambos trabajos más relevantes para la distribución. De todas maneras, existen aun líneas de trabajo no finalizadas que completarán ambos trabajos.

Este apartado se estructura de la siguiente manera. En primer lugar, se presenta la forma en que se introduce la reflectividad estructural en la máquina abstracta, tanto

desde un punto de vista interno, de la implementación, como desde un punto de vista externo, lo que es visible al programador.

En segundo lugar, se hace lo propio con la reflectividad del comportamiento.

Finalmente, se muestra la forma en que se ha implementado el mecanismo de invocación remota, como característica fundamental del sistema de distribución.

16.5.1 Clases primitivas para la reflectividad estructural

La reflectividad estructural, tal y como se describió en el capítulo 9, hace accesibles y manipulables al usuario los objetos del sistema. Es necesario, por tanto, definir en el lenguaje ensamblador de la máquina un conjunto de clases básicas que conforman el módulo de reflectividad estructural: el área de clases, de instancias, de referencias, y de hilos y las clases, las instancias, los métodos y las instrucciones. Dichas clases se describen a continuación.

16.5.1.1 Exposición de los elementos que forman la arquitectura de la máquina

Clase ClassArea

La clase ClassArea representa o expone el área de clases de la máquina abstracta, permitiendo su acceso y modificación. Su interfaz es la siguiente:

```
Class ClassArea
Isa object
Methods

   GetClasses():array;
   IsaClass(string): bool;
   GetClass(string): _Class;
   NewClass(string): _Class;
   DupClass(string, string): _Class;

Endclass
```

El método **GetClasses** obtiene del área de clases los identificadores de todas las clases almacenadas en el área de clases. Estos nombres se almacenan en un array de objetos de tipo _Class para su posterior utilización.

Por su parte **GetClass** obtiene un objeto de tipo _Class que representa a la clase cuyo nombre se pasa como parámetros. Esto hace que las clases sean accesibles y manipulables mediante instrucciones normales de la máquina, como se verá más adelante.

IsaClass interroga al área de clases acerca de la existencia de una clase con el nombre que se pasa como parámetro. El retorno de un valor cierto, supondrá la existencia de tal clase. En caso de retornar falso, tal clase no está definida.

El método **NewClass**, almacena en el área de clases una nueva clase cuya referencia se pasa como parámetro. Y por último, el método **DupClass** duplica una clase ya existente, que tenga como nombre la cadena pasada como primer parámetro, creando una segunda clase, que tiene como nombre el segundo parámetro pero que se coloca como descendiente directa de la clase origen en la jerarquía de clases. Posteriormente, gracias a la exposición de las clases, podremos modificar cualquiera de las clases creadas a voluntad.

Clase InstanceArea

La clase InstanceArea representa o expone el área de instancias de la máquina abstracta, permitiendo su acceso y modificación. Su interfaz es la siguiente:

```
Class InstanceArea
Isa object
Methods

   GetInstances():array;
   NewInstance(string):instance;
   InstanceofthisRef(reference):instance;
   SetLoadbyUser(string, integer);
   GetLoadbyUser(string):integer;
   SerializeInstance(object): string;
   CreateObjectFromString(string): object;

Endclass
```

El método **GetInstances** obtiene referencias a todas las instancias almacenadas en el área de instancias que, a su vez, se almacenarán en un array para su posterior utilización.

Por su parte **NewInstance** crea una nueva instancia del tipo indicado como parámetro. Devuelve una referencia a un objeto de tipo Instance que podrá manipularse con las instrucciones normales de la máquina.

El método **InstanceofThisRef** inspecciona el área de instancias buscando el objeto ligado a la referencia que se pasa como parámetro. Como resultado, crea un objeto de tipo instancia para reflejar las características de la instancia hallada.

Los métodos **GetLoadbyUser** y **SetLoadbyUser** permiten averiguar y establecer, respectivamente, la carga de objetos que el usuario identificado tiene permitido. Su principal utilidad radica en el caso de que se eleven demasiadas excepciones por falta de espacio en el área de instancias, en cuyo caso, pueden establecerse límites a la creación de instancias.

Por último, los métodos **SerializeInstance** y **CreateObjectFromString** son utilizados por los objetos movedor con el fin de realizar la migración de objetos. El primero de ellos obtiene una cadena de caracteres que representa al objeto referenciado en el parámetro. El segundo método crea una instancia en el área de instancias saltándose el procedimiento habitual, dado que se necesita que el objeto conserve su identificador.

La exposición de las instancias permite la creación y modificación de las mismas en tiempo de ejecución lo que resultará particularmente útil para implantar la reflectividad del comportamiento.

Clase ReferenceArea

La clase ReferenceArea representa o expone el área de referencias de la máquina abstracta, permitiendo su acceso y modificación. Su interfaz es la siguiente:

```
Class ReferenceArea
Isa object
Methods

     NewReference(string):reference;
     SetLoadbyUser(string, integer);
     GetLoadbyUser(string):integer;

Endclass
```

El método **NewReference** permite crear una referencia nueva en el área de referencias. El parámetro que se pasa es el nombre de la clase.

Por último, los métodos **GetLoadbyUser** y **SetLoadbyUser** permiten averiguar y establecer, respectivamente, la carga de referencias que el usuario identificado tiene permitido. Similares a los métodos del mismo nombre de la clase InstanceArea.

Clase ThreadArea

El área de hilos permite el acceso y modificación del área del mismo nombre definida en la arquitectura de referencia de la máquina, en el capítulo 7. Su interfaz es:

```
Class ThreadArea
Isa object
Methods

     HowMany():integer;
     MaxLoad(integer);

Endclass
```

El método **HowMany** devuelve el número de hilos en el área. Mientras que **MaxLoad** permite establecer un límite superior al número de hilos **activos** en la máquina, con el fin de distribuir la carga en varias máquinas, si fuese necesario.

16.5.1.2 Exposición de los elementos estructurales de la implantación en tiempo de ejecución de los objetos

Clase _Class

Esta clase, que expone precisamente diversos aspectos de la representación en tiempo de ejecución de las clases, permite el acceso y modificación de cualquier clase definida en el sistema. Su interfaz es la siguiente:

```
Class _Class
Isa object
Methods

   GetName():string;
   GetInterfaz():array;
   GetParent():array;
   GetHierarchy():array;
   GetAggregates():array;
   GetAssociates():array;
   SetHierarchy(array);
   SetAggregates(array);
   SetAssociates(array);
   GetMethod(string):Method;

Endclass
```

El conjunto de métodos de la clase _Class está dividido en aquellos que permiten **interrogar a la clase** acerca de su estructura y aquellos que permiten **actuar** sobre la misma, modificándola en tiempo de ejecución.

Así entre los primeros se encuentran los siguientes: **GetName** permite interrogar a una clase acerca del nombre asignado en la definición. Obviamente, devuelve ese nombre como una cadena de caracteres. **GetInterfaz** obtiene un array con los nombres del conjunto de métodos que la clase interrogada implementa. **GetParent** inspecciona la clase en busca de los antecesores directos en la jerarquía de clases, mientras **GetHierarchy** realiza una inspección completa de la cadena de herencia de la clase. **GetAggregates** y **GetAssociates** obtienen las clases agregadas y asociadas, respectivamente, y por último, el método **GetMethod** permite obtener la descripción de un método de la clase a partir de una cadena de caracteres con su nombre.

Y entre los segundos, aquellos que permiten definir o modificar los elementos que constituyen la clase: **SetHierarchy**, **SetAggregates** o **SetAssociates**.

Clase Instance

Representa una instancia en tiempo de ejecución. Su interfaz consta de los siguientes métodos:

```
Class Instance
Isa object
Methods

   GetClass():string;
   GetExecArea():ExecObjectArea;
   GetMetaSpace():MetaSpace;
   GetAggregates():array;
   GetAssociates():array;

Endclass
```

Todos sus métodos se dirigen a realizar introspección sobre la instancia en tiempo de ejecución.

El método **GetClass** devuelve el nombre de la clase de la que es instancia, **GetExecArea** devuelve un objeto de tipo **ExecObjectArea** que, como se verá representa el

área de ejecución de cada objeto, **GetMetaSpace**, devuelve un objeto de tipo **MetaSpace** que engloba al conjunto de meta-objetos que componen el entorno de un objeto base. Este método es el método básico para permitir el acceso y manipulación del meta-espacio de un objeto en tiempo de ejecución.

Por último, **GetAggregates** y **GetAssociates** devuelve un array que contiene, respectivamente, los agregados y los asociados de la instancia.

Clase Method

La clase Method expone los métodos de una clase y permite el acceso y modificación a los mismos. Permite crear métodos en tiempo de ejecución y es fundamental para poder definir clases en tiempo de ejecución. Su interfaz es:

```
Class Method
Isa object
Methods

    GetName():string;
    GetInstructionsSet():array;
    GetReturnType():string;
    GetReferences():array;
    GetInstances():array;
    GetParams():array;
    SetInstructionsSet(array);
    SetReturnType(string);
    SetReferences(array);
    SetInstances(array);
    SetParams(array);

Endclass
```

Nuevamente, los métodos están divididos en dos conjuntos simétricos. El primero, compuesto por los métodos **GetName**, **GetInstructionsSet**, **GetReturnType**, **GetReferences**, **GetInstances** y **GetParams**, permite inspeccionar un método obteniendo una cadena de caracteres con su nombre, un array de instrucciones que constituyen el cuerpo del método, el tipo del valor de retorno, si existe, y las referencias, instancias y parámetros que el método recibe, respectivamente.

El segundo conjunto, formado por **SetInstructionsSet**, **SetReturnType**, **SetReferences**, **SetInstances** y **SetParams**, permite dar valor y modificar el conjunto de instrucciones de un método, su valor de retorno, referencias, instancias y parámetros, respectivamente.

Clase Instruction

La clase Instruction permite la creación de nuevos objetos instrucción de alguno de los tipos de instrucción definidos. Es imprescindible si se pretenden definir nuevas clases en tiempo de ejecución y su interfaz es la siguiente:

```
Class Instruction
Isa object
Methods

   NewAssign(string, string);
   NewCall(string, string, array, string);
   NewDelete(string);
   NewExit();
   NewHandler(integer);
   NewNew(string);
   NewJmp(integer);
   NewJT(string, integer);
   NewJF(string, integer);
   NewJNull(string, integer);
   NewJNNull(string, integer);
   NewSend(string, string, array, string);
   NewThrow();

Endclass
```

Los métodos se refieren todos a la creación de instrucciones de los tipos que hay definidos en el procesador recibiendo como parámetros los argumentos que necesitan tales instrucciones.

Así, **NewAssign**, crea una instrucción de asignación y necesita los nombres de las referencias que entran en juego en la asignación.

NewCall, se refiere a una instrucción de invocación síncrona. Sus parámetros son el nombre de la referencia, el nombre del método, los parámetros del método y la referencia de retorno, si existe.

NewDelete, crea una instancia de la instrucción Delete, que borra la referencia que tenga como nombre el argumento enviado.

NewExit crea una instrucción de finalización.

NewHandler, crea una instrucción de tipo Handler y recibe como parámetro el desplazamiento, dentro del cuerpo del método, donde salta el control de flujo.

NewNew, crea un objeto de tipo Instrucción que representa la instrucción New. Obviamente, como parámetro recibe el nombre de la referencia.

NewJmp, crea una instrucción de salto incondicional. El parámetro representa el offset donde continúa el flujo de ejecución.

NewJT, **NewJF**, **NewJNull** y **NewJNNull**, crean instrucciones de salto condicional. Reciben como parámetro, en primer lugar el nombre de la referencia a inspeccionar y en segundo lugar, el desplazamiento donde continúa el flujo de ejecución, en caso de que la instrucción tome valor cierto.

NewSend, al igual que NewCall, es una instrucción de paso de mensajes, pero, en este caso, asíncrona. Los parámetros son los mismos, ya que solo cambia la semántica de la instrucción.

Por último, **NewThrow** crea una instrucción de emisión de una excepción. No lleva parámetros.

16.5.1.3 Exposición del motor en tiempo de ejecución

Clase ExecObjectArea

La clase ExecObjectArea representa o expone el área de ejecución del objeto, es decir, permite inspeccionar qué métodos se están ejecutando en dicho objeto. Esto, junto con la exposición del objeto interno hilo que se verá a continuación, constituyen una herramienta imprescindible para poder definir la planificación en el meta-nivel, a la vez que suponen una gran ayuda para las tareas de depuración.

Su interfaz es la siguiente:

```
Class ExecObjectArea
Isa object
Methods

   GetThreads():array;
   GetLoad():integer;
   SetLoad():integer;

Endclass
```

El método **GetThreads** obtiene referencias a objetos de tipo hilo que representan todos los métodos en ejecución en el objeto.

Clase Thread

La clase Thread representa o expone la ejecución de cada uno de los métodos de un objeto. Permite inspeccionar el estado del hilo, la cadena de llamadas que rodean al hilo, hacia delante y hacia atrás, así como realizar acciones como suspender el hilo actual, reanudar un hilo, etc.

Su interfaz es la siguiente:

```
Class Thread
Isa object
Methods

    GetState():string;
    GetContext():context;
    GetPreviousThread():Thread;
    GetNextThread():Thread;
    Kill();
    Suspend();
    Resume();
    Start();

Endclass
```

El método **GetState** informa acerca del estado actual del hilo referenciado.

GetContext permite obtener el contexto de ejecución actual del hilo. Junto con la clase Context permite navegar por la pila de ejecuciones de métodos en el objeto al que está asociado. Si a ello se unen **GetPreviousThread** y **GetNextThread** que permiten obtener referencias a objetos de tipo hilo que representan el hilo anterior y siguiente, se puede seguir sin problema, toda la cadena de llamadas a métodos que incluye al presente hilo.

Por último, los métodos **Kill**, **Suspend**, **Resume** y **Start** permiten actuar sobre el objeto interno de la máquina y son básicos para la construcción de planificadores que, de otra forma no podrían actuar sobre los hilos de la máquina. Se ocupan de matar, suspender, reanudar y comenzar la ejecución de un hilo, respectivamente.

Clase Context

Permite el acceso a un contexto de ejecución de la máquina, aunque no sea el contexto activo actualmente. Los métodos de su interfaz son:

```
Class Context
Isa object
Methods

   GetPreviousContext():Context;
   GetNextContext():Context;
   GetInstances():array;
   GetReferences():array;
   GetMethod():string;

Endclass
```

GetPreviousContext y **GetNextContext** permiten explorar la pila de contextos de un hilo moviéndose hacia delante y hacia atrás en la misma.

El resto de los métodos permiten explorar un contexto determinado obteniendo las referencias e instancias con los métodos **GetInstances** y **GetReferences** respectivamente. Ambos devuelven arrays con las referencias a instancias y referencias del contexto.

Por último, **GetMethod** identifica el método en ejecución en el contexto dado.

16.5.2 Representación interna de la reflectividad estructural

Esta jerarquía de clases primitivas permite al programador acceder y modificar al estado interno del sistema en tiempo de ejecución.

Esto debe verse reflejado en la implantación interna mediante un conjunto de clases que representen los elementos estructurales del lenguaje Carbayón que son expuestos de manera reflectiva (los descritos en el apartado anterior).

Al igual que en el prototipo no reflectivo, esto se traduce en la ampliación de las distintas jerarquías de clases del prototipo que se mostraron para aquel caso.

16.5.2.1 Jerarquía de clases para representar internamente las clases

La implantación de una máquina dotada de reflectividad estructural se lleva a cabo mediante la exposición de los objetos del nivel base.

Para ello, se definen un conjunto de clases que exponen los aspectos de implantación de esos objetos, tal como se explicó en el capítulo 9.

En la figura siguiente se ofrece una visión de la jerarquía de clases.

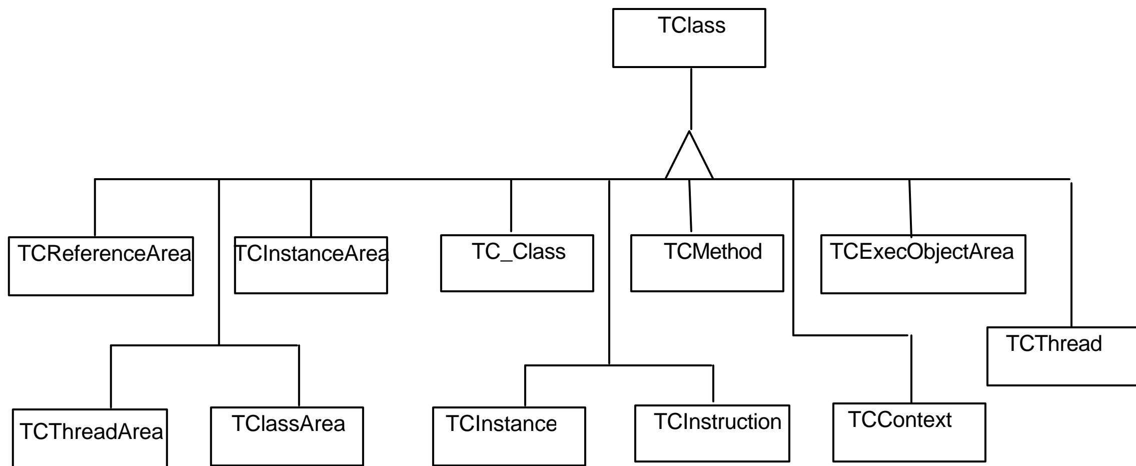


Figura 16.8. Jerarquía de clases para la reflectividad estructural en Carbayonia

Este conjunto de clases permite al sistema acceder a los datos en tiempo de ejecución, realizando introspección e incluso intercesión sobre él.

Esto último es fundamental ya que, dado que el meta-sistema no es más que un reflejo del sistema, la actuación sobre él se traduce en modificaciones similares en el sistema base reflejado.

16.5.2.2 Jerarquía de clases de las instancias

A continuación se muestra la jerarquía de clases del prototipo para las instancias primitivas del módulo de reflectividad. La raíz de la jerarquía es la clase abstracta TInstance, de la cual heredan todas las clases que representan instancias, tanto las que aquí se describen como las instancias de clases primitivas o de usuario.

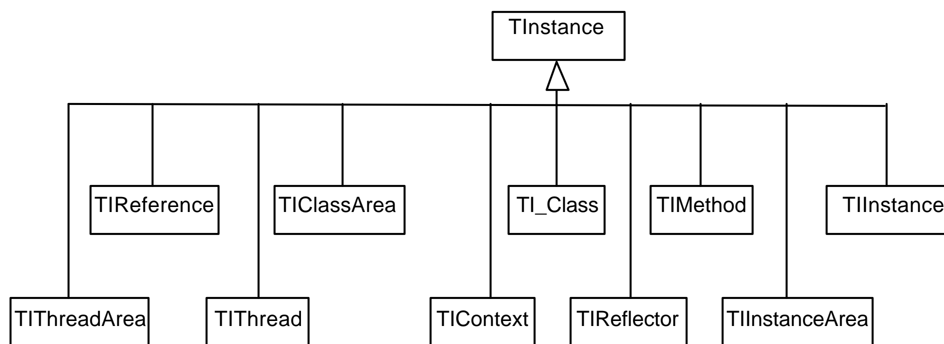


Figura 16.9. Jerarquía de clases de las instancias para la reflectividad estructural.

Por ejemplo, la clase TI_Class es una clase concreta derivada de TInstance que representa una instancia de la clase primitiva _Class, al igual que la clase TIInteger, también derivada de TInstance, representaba una instancia de la clase primitiva Integer.

16.5.2.3 Jerarquía de clases de los métodos

En la siguiente figura se muestra la jerarquía de clases del prototipo para los métodos del módulo de reflectividad. La raíz de la jerarquía es la clase abstracta TMethod, que define las operaciones que debe ofrecer un método del simulador en

tiempo de ejecución. La funcionalidad de cada una de estas operaciones tiene que darla cada una de sus clases derivadas de acuerdo a sus características.

Simplemente se representan algunas de las clases, lo cual es suficiente para dar una idea de la jerarquía.

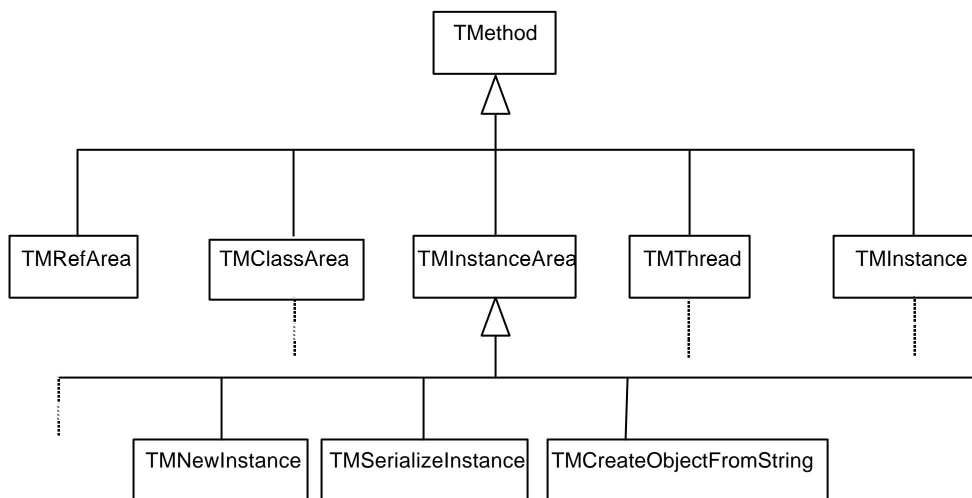


Figura 16.10. Jerarquía de clases de los métodos para la reflectividad estructural.

Por ejemplo, la clase TMSerializeInstance es una clase concreta derivada de TMInstanceArea que representa el método SerializeInstance de la clase primitiva InstanceArea.

16.5.2.4 Jerarquía de clases de las instrucciones

Finalmente, se muestra la jerarquía de clases del prototipo para las instrucciones del módulo de reflectividad. La raíz de la jerarquía es la clase abstracta TInstruction, que define las operaciones que debe ofrecer una instrucción del simulador en tiempo de ejecución. La funcionalidad de cada una de ellas deberá implementarse en las clases derivadas de acuerdo a las instrucciones que representen.

Por simplicidad, se representan únicamente algunas de las clases, lo cual es suficiente para dar una idea de la jerarquía.

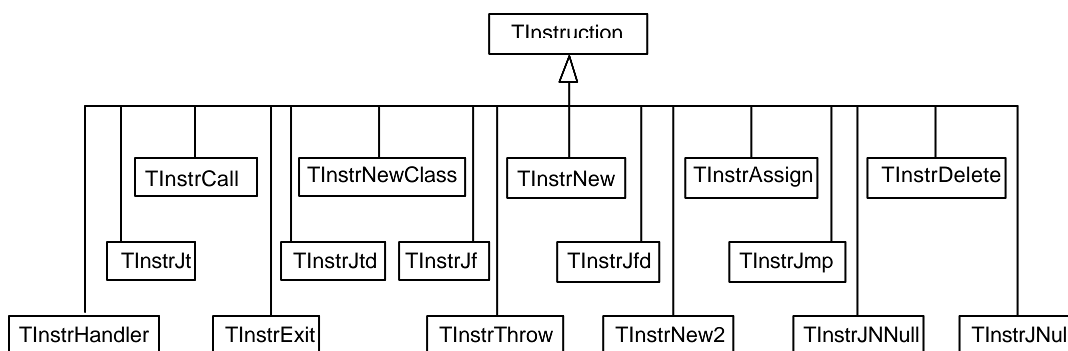


Figura 16.11. Jerarquía de clases de las instrucciones para la reflectividad estructural

16.5.3 Clases primitivas para la reflectividad del comportamiento: implantación del sistema operativo

A continuación se muestra el conjunto de clases primitivas que permite al sistema acceder a los datos en tiempo de ejecución acerca del comportamiento particular que la máquina ofrece para él. La modificación de este comportamiento se lleva a cabo a través de la ampliación de esta jerarquía y de la modificación del objeto MetaSpace del objeto base.

La modificación de la jerarquía puede llevarse a cabo tanto añadiendo clases en tiempo de compilación como en tiempo de ejecución, gracias a la reflectividad estructural que expone el objeto clase de la máquina abstracta y la exposición del área de clases de la máquina.

16.5.3.1 Clase MetaSpace

La clase MetaSpace representa los objetos del meta-espacio en tiempo de ejecución, permitiendo conocer qué meta-objetos lo componen y modificarlos, logrando así adaptación del entorno en tiempo de ejecución.

La Clase MetaSpace tiene una interfaz muy simple.

```
Class MetaSpace
Isa Object
Methods

   GetMetaSpace(): array;
   GetMObyClass(string):MetaObject;
   AttachMObyClass(string, MetaObject);

Endclass
```

El método **GetMetaSpace** devuelve un array de meta-objetos que componen el meta espacio. Posteriormente, se podrán inspeccionar estos meta-objetos.

El método **GetMObyClass** devuelve el meta-objeto de la clase especificada, mientras **AttachMObyClass** especifica un nuevo meta-objeto para exponer determinada característica del entorno del objeto.

16.5.3.2 La Clase MetaObject

La clase MetaObject representa los meta-objetos en tiempo de ejecución. Se expresa en una jerarquía aparte de los objetos de usuario, aunque se codifican en Carbayón y pueden – y deben – ser definidos por el usuario.

La Clase MetaObject tiene una interfaz muy simple – igual que la clase object – y cada subclase de la misma especializará su comportamiento y creará una jerarquía independiente.

```
Class MetaObject
Isa Object
Methods

   GetClass(): string;
   GetID():integer;
   Isa(string):bool;
   GetObjectBase():array;

Endclass
```

El método **getClass** devuelve una cadena de caracteres que contiene el nombre de la clase a la que pertenece el objeto.

El método **getID** devuelve una instancia de un entero que identifica de forma única al objeto. De esta forma se puede averiguar si se tiene dos referencias apuntando al mismo objeto (aunque estas sean de distinto tipo).

El tercer método –**isa**– devuelve una instancia de un objeto de tipo bool que indica si la instancia pertenece a esa clase o a una derivada de ella.

Por último, **GetObjectBase** devuelve una referencia a un array de objetos de cuyo meta-espacio forma parte.

A continuación se ofrece la jerarquía básica de meta-objetos. Cada usuario puede, no sólo modificar los métodos de la interfaz, sino también añadir nuevos métodos, ampliando de esta forma su jerarquía.

16.5.3.3 Clase Emisor

La clase Emisor representa los objetos del meta-espacio que se encargan del envío de mensajes en tiempo de ejecución.

La interfaz base de esta clase es tan simple como sigue:

```
Class MOEmisor
Isa MetaObject
Methods

   RCallE(string, string, array): string;
   RSendE(string, string, array): string;

Endclass
```

El método **RCallE** define las operaciones de envío síncrono de mensajes, mientras **RSendE** se refiere al envío asíncrono. En ambos casos, los parámetros que reciben son el nombre de la referencia del objeto destino, el nombre del método a invocar, un array con los argumentos y el nombre de la referencia donde se guarda el resultado, si existe.

Estos métodos retorna diversos resultados, en función de cómo esté configurado el meta-objeto destino. Estos resultados puede utilizarse para ampliar la semántica del meta-objeto.

Un ejemplo de meta-objeto Emisor más sofisticado sería el siguiente:

```
Class MOEmisorComplex
Isa MOEmisor
Methods

   RCallE(string, string, array, integer): string;
   MethodTimeOut();
   NoFindDest();

Endclass
```

Este meta-objeto redefine el meta-objeto anterior modificando el método **RCallE** de tal forma que, entre las operaciones de envío de mensajes, se instale una alarma que controle el tiempo de ejecución de un método, por ejemplo para evitar largos tiempos de espera en una invocación remota que debe atravesar una red muy saturada. También define un comportamiento excepcional en caso de que no se encuentre el objeto destino. Los parámetros que recibe son los mismos, si exceptuamos el número de segundos que se intenta la ejecución del método.

El método **MethodTimeOut** define las acciones a realizar en caso de que el objeto destino no sea capaz de ejecutar el método solicitado en el tiempo especificado.

El método **NoFindDest** especifica las acciones a realizar en caso de que no se encuentre el objeto destino. La acción por defecto es elevar una excepción. Sin embargo, puede intentar solucionarse buscando un servidor alternativo.

16.5.3.4 Clase Receptor

La clase Receptor representa los objetos del meta-espacio que se encargan de la recepción de mensajes en tiempo de ejecución.

La interfaz base de esta clase es tan simple como sigue:

```
Class MOReceptor
Isa MetaObject
Methods

   RCallR(string, string, array): string;
   RSendR(string, string, array): string;

Endclass
```

El método **RCallR** define las operaciones de recepción síncrona de mensajes, mientras **RSendR** se ocupa de la recepción asíncrona. Los parámetros que reciben son el nombre de la referencia del objeto origen, el nombre del método a invocar, un array con los argumentos y el nombre de la referencia donde se guarda el resultado, si existe.

Este método retorna diversos resultados, en función de cómo esté configurado el meta-objeto destino y los retornará al meta-objeto Emisor.

Un ejemplo de meta-objeto Receptor más sofisticado sería el siguiente:

```
Class MOReceptorComplex
Isa MOReceptor
Methods

   RCallR(string, string, array): string;
   OverLoad(string, string, array, string): object;
   DelegateMethod(string, string, array, string): object;
   DelayMehod();

Endclass
```

Este meta-objeto redefine el meta-objeto anterior modificando el método **RCallR** de tal forma que, entre las operaciones de recepción de mensajes, se estudie el entorno en tiempo de ejecución del objeto base tomándose decisiones acerca de la posibilidad de ejecución del mismo en función de la carga.

OverLoad define las acciones a realizar en caso de carga excesiva, por defecto el método RCallR rechazaría el mensaje. Los parámetros son, nuevamente, el nombre de la referencia a quien delegar, el nombre del método, los argumentos y el nombre de la referencia de retorno. Así mismo, devuelve un resultado que dependerá de si todo ha ido bien o si se ha producido algún problema.

Otras acciones pueden ser delegarlo en otro objeto (posiblemente una réplica), **DelegateMethod** o retrasar su ejecución, **DelayMethod**.

16.5.3.5 Clase Sincronizador

La clase Sincronizador representa los objetos del meta-espacio que se encargan de definir el comportamiento del objeto ante la invocación de métodos.

La interfaz base de esta clase es tan simple como sigue:

```
Class MOSynchronizer
Isa MetaObject
Methods

   ExecNewMethod(string, string): string;
   EndMethod(string);
   RestartMethod(string);
   StopMethod(string);

Endclass
```

El método **ExecNewMethod** indica las acciones a realizar cuando se solicita la ejecución de un método. Los parámetros corresponden al origen de la llamada y al nombre del método, con el fin de evitar bloqueos mutuos.

Los métodos **EndMethod** y **StopMethod** definen las operaciones que se llevan a cabo para determinar si otros métodos pueden pasar a ejecutarse como consecuencia de la finalización o suspensión del método cuyo nombre se pasa como parámetro.

De forma similar el método **RestartMethod** implica la reanudación de la ejecución del método, lo que puede implicar que otros cambien su estado.

La acción por defecto es la más liberal posible lo que implica que cualquier método puede ejecutarse siempre, dejando en manos del programador la sincronización adecuada. Los métodos estarían vacíos, excepto por la llamada al planificador correspondiente para advertirle de la presencia de un nuevo hilo de ejecución.

Pero cada usuario puede redefinir estos métodos aplicando la política de sincronización que desee.

Un ejemplo de Sincronizador más sofisticado, sería aquel que definiese una actuación distinta para cada método:

```
Class MOSynchronizerComplex
Isa MetaObject
Methods

   ExecNewMethod(string, string): object;
   EndMethod(string);
   RestartMethod(string);
   StopMethod(string);
   EntryM();
   ExitM();
   StopM();
   ResumeM();
   IentryM();
   IexitM();
   IstopM();
   IresumeM();

Endclass
```

Donde los métodos EntryM, ExitM, StopM, ResumeM, IentryM, IexitM, IstopM, IresumeM estarían definidos para todos o parte de los métodos del objeto del nivel base.

16.5.3.6 Clase Scheduler

La clase Scheduler representa los objetos del meta-espacio que se encargan de definir la planificación del entorno.

La interfaz base de la clase planificador universal es tan simple como sigue:

```
Class MOUniversalScheduler
Aggregation queue
Isa MetaObject
Methods

   ScheduleNext();
   Enqueue(Thread);
   IsEmpty():Bool;
   GetQueue():queue;

Endclass
```

El método **ScheduleNext** indica a la máquina abstracta la referencia al hilo del que debe ejecutar instrucciones.

Enqueue inserta en la cola de hilos un nuevo hilo en la posición indicada. Y, por último, **IsEmpty** inspecciona la cola de hilos para determinar si hay alguno y **GetQueue** retorna una cola con los hilos que este planificador tenga definidos.

16.5.3.7 Clase localizador

La clase Localizador representa los objetos del meta-espacio que se encargan de implementar la política de localización de objetos.

La interfaz base de la clase localizador es como sigue:

```
Class MOLocalizador
Aggregation CacheInstanciasNoLocales: ArrayRefs;
Isa MetaObject
Methods

   Find(Object, LocationOption): Node;
   RegisterInCache(Object, Node);

Endclass
```

El método **Find** permite obtener la ubicación actual de un objeto dada una referencia al mismo. La información de ubicación queda encapsulada en un objeto de tipo **Node**. Adicionalmente, se pueden proporcionar opciones de localización que permiten modificar el comportamiento por defecto del método, tal y como se explicó en el capítulo 12.

El método **RegisterInCache**, por su parte, se utiliza para almacenar una correspondencia entre un objeto y su ubicación actual, de tal forma que pueda ser utilizada en posteriores invocaciones al método **Find**.

16.5.3.8 Clase movedor

La clase Movedor[21] representa los objetos del meta-espacio que se encargan de implementar la política de migración de objetos.

La interfaz base de la clase movedor es como sigue:

```
Class MOMovedor
Isa MetaObject
Methods

   Move(Object, Node);

Endclass
```

El método **Move** tiene una interfaz muy simple. Únicamente necesita una referencia al objeto a mover y el nodo hacia el que se va a realizar el movimiento. Toda la complejidad de la operación de migración, explicada en el capítulo 12, está encapsulada en dicho método, con las necesarias invocaciones a otros objetos que colaboran en la operación.

16.5.3.9 Clase comunicador

La clase Comunicador representa los objetos del meta-espacio que se encargan de implementar la comunicación a través de la red entre diferentes nodos.

La interfaz base de la clase comunicador es como sigue:

[21] Este meta-objeto se encuentra actualmente en desarrollo en el prototipo.

```
Class MOComunicador
Isa MetaObject
Methods

   Send(Message, Node);
   MsgReceived(Message);

Endclass
```

El método **Send** es utilizado para enviar un mensaje a un nodo. El contenido del mensaje habrá sido construido por algún otro objeto, y es indiferente para el meta-objeto comunicador. Como ya se ha comentado, existe la posibilidad de que incluso vaya cifrado.

El método **MsgReceived** es invocado cada vez que llega un mensaje a través de la red, a la manera de un gestor de eventos o interrupciones. El meta-objeto comunicador entregará, en general, el mensaje a un meta-objeto trabajador que sabrá qué tiene que hacer con él.

16.5.4 Representación interna de la reflectividad del comportamiento

Además de exponer la arquitectura de la máquina y la estructura de los objetos del nivel base, también se exponen partes del comportamiento de la máquina mediante una jerarquía de clases similar a la anterior y que permite la modificación del comportamiento de la misma, adaptándolo a cada aplicación.

Los aspectos del comportamiento de los objetos que se exponen son la invocación de métodos y la migración (que conforman la distribución de objetos), la sincronización interna de los objetos (necesaria para la migración) y la planificación.

A diferencia de la jerarquía anterior, esta nueva jerarquía se corresponde con la funcionalidad del sistema operativo. Sin embargo, comparte con ella la forma en que se le da soporte en el prototipo, que consiste en ampliar las jerarquías de TClass, TInstance y TMethod, de la forma que ya se hizo para la reflectividad estructural. Debido a esta analogía, se mostrará únicamente de qué manera se amplía la jerarquía de TClass.

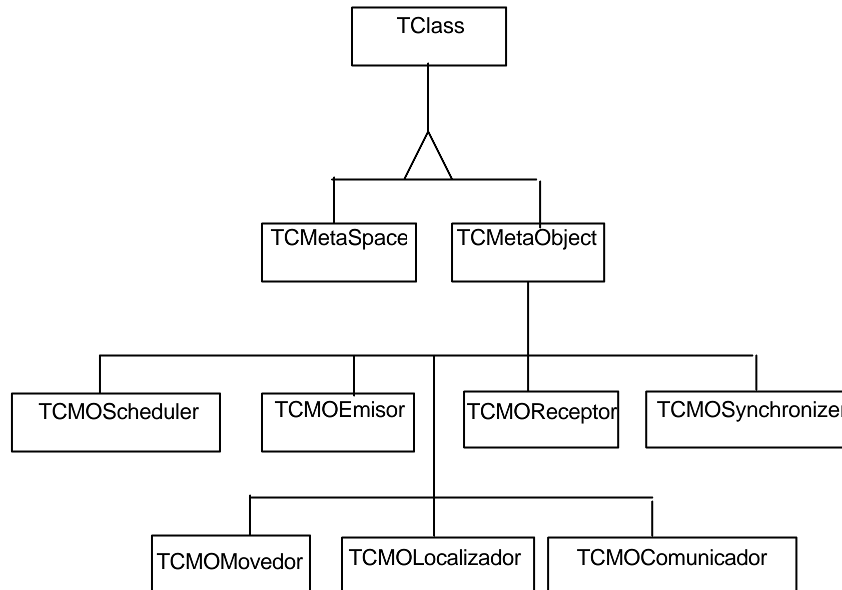


Figura 16.12. Jerarquía de clases para la reflectividad del comportamiento.

16.5.5 Descripción dinámica del prototipo

Dado que la invocación de métodos es la operación fundamental de todo sistema de gestión de objetos, se ilustra a continuación su funcionamiento dentro del prototipo del sistema integral distribuido en la forma de diagramas de colaboración.

Como ya se ha mencionado en repetidas ocasiones, en el sistema integral distribuido se distinguen dos tipos de invocación: local y remota.

La invocación local es resuelta íntegramente por la máquina abstracta. Así, es posible describirla con diagramas de colaboración que utilizan únicamente objetos de la implementación de la máquina (escritos en su lenguaje de implementación, C++) y los mensajes que se intercambian.

Sin embargo, la invocación remota necesita de la colaboración de objetos del sistema operativo (escritos en el lenguaje ensamblador de la máquina, Carbayón) que posibilitan la interacción entre máquinas abstractas distintas. Por ello, los objetos que se representan en los diagramas de colaboración que describen la invocación remota son objetos definidos por el lenguaje de la máquina y no de su implementación.

16.5.5.1 Invocación local de un método de usuario

El escenario que se describe a continuación se produce cuando se ejecuta una instrucción call, lo que se traduce en la invocación del método exec en el objeto TInstrCall, encargado de realizar las llamadas a los métodos.

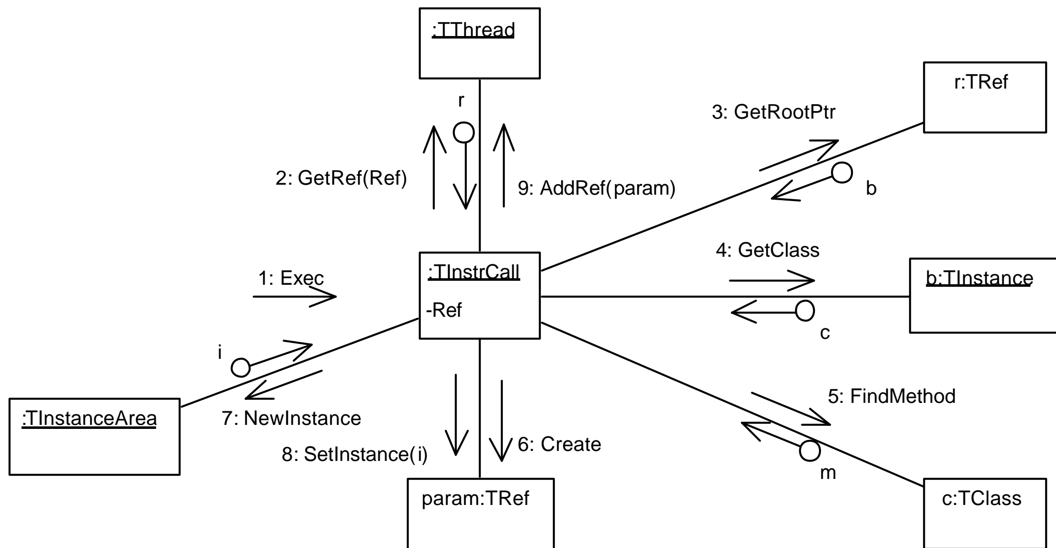


Figura 16.13. Ejecución de la instrucción call.

Cuando el objeto TInstrCall recibe la petición de ejecutarse solicita al Thread sobre el que se ejecuta que le dé acceso a la referencia sobre la que se ha realizado la invocación.

Una vez obtenida la referencia, se le solicita la dirección de la instancia a la que apunta. Si dicha instancia no es una instancia de usuario, o pertenece a un metaobjeto, efectua los siguientes pasos:

- Se le pide a la instancia que identifique la clase a la que pertenece para de esta manera solicitarle un puntero al método requerido (paso 5).
- Finalmente se comunica al método que un hilo le ha invocado, mediante el mensaje invokeMethod.

En los pasos 6, 7 y 8 se crean las referencias y sus instancias asociadas, de cada uno de los parámetros que se van a pasar al meta-objeto Emisor de la instancia en la que se realiza la llamada (instancia a la que se invoca, nombre del método, valor de retorno y parámetros).

Las referencias creadas se van añadiento al contexto de ejecución del hilo (referencias locales), para el posterior acceso a ellas (paso 9).

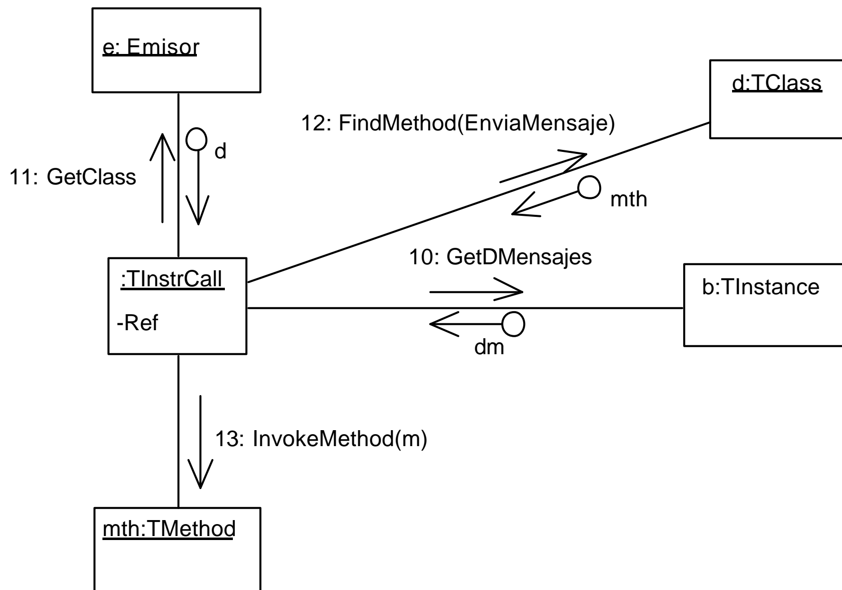


Figura 16.14. Ejecución de la instrucción call (continuación).

A continuación, se obtiene el objeto Emisor de la instancia actual (la que realiza la llamada) y se solicita la clase a la que pertenece, para obtener un puntero al método EnviaMensaje (pasos 11 y 12).

Finalmente se envía el mensaje al objeto Emisor, mediante la invocación del método obtenido en el paso anterior.

Llamada al meta-objeto Receptor

Este escenario representa el proceso de recepción por parte del meta-objeto Receptor del mensaje que debe enviar a la instancia cuyo método se quiere invocar.

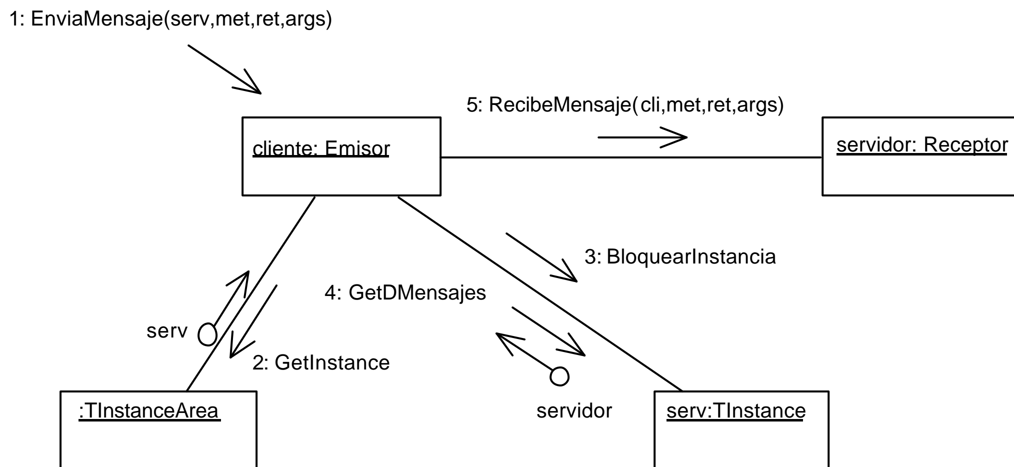


Figura 16.15. Envío de un mensaje al objeto Receptor.

Lo primero que hace el objeto Emisor una vez recibido el mensaje a enviar, es obtener un puntero a la instancia a la que hay que enviarle el mensaje (instancia servidor).

Si la llamada al método que se va a realizar va a devolver un objeto de retorno, se bloquea dicho objeto (instancia), para que así no se pueda acceder a él hasta que el método lo haya devuelto (paso 3).

Después se obtiene el objeto Receptor de la instancia servidor (paso 4), y se le pasa el mensaje mediante la llamada a su método RecibeMensaje (paso 5).

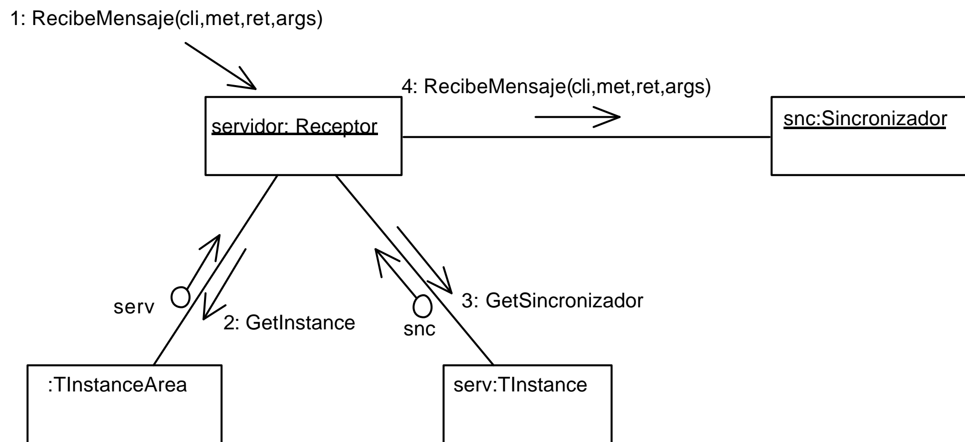


Figura 16.16. Recepción de un mensaje por el objeto Receptor.

El objeto Receptor procede de la forma siguiente. En primer lugar, obtiene un puntero a la propia instancia servidora que va a recibir el mensaje (y a la que está asociado dicho objeto Receptor). Después obtiene el objeto Sincronizador de dicha instancia (paso 3).

Finalmente le envía el mensaje al objeto Sincronizador, mediante la llamada a su método RecibeMensaje (paso 4). El objeto Sincronizador será el responsable último de poner el método en ejecución.

16.5.5.2 Invocación remota de un método de usuario

El escenario que se describe a continuación se produce cuando el objeto que realiza la invocación de un método y el objeto invocado no están ubicados en el mismo nodo. Como ya se comentó anteriormente, el escenario que se muestra describe las interacciones entre objetos de nivel usuario, y no de la implementación, dado que las invocaciones remotas no son realizadas por la máquina abstracta, sino por objetos del sistema operativo. Cada una de las invocaciones mostradas en el escenario siguiente se corresponde con el escenario mostrado para la invocación local de objetos de usuario del apartado anterior. Dado que no es preciso tal detalle, se ha omitido por razones de claridad.

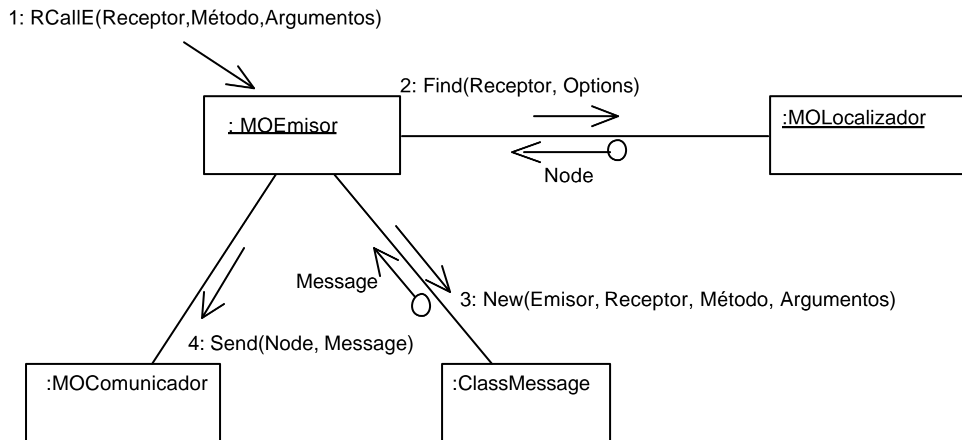


Figura 16.17. Parte local de una invocación remota.

En primer lugar, el meta-objeto emisor del objeto cliente obtiene el control de manera reflectiva (paso 1) por parte de la máquina abstracta. El meta-objeto emisor será el responsable de realizar la parte local de la invocación remota. Este meta-objeto obtiene la información de ubicación del objeto a invocar, a través del meta-objeto localizador (paso 2).

Una vez obtenida la dirección del nodo en que reside el objeto servidor, pasa a crear el mensaje que contendrá todos los datos de la invocación (paso 3) y se lo pasa al objeto comunicador (paso 4), que lo envía a través de la red hasta el nodo destino.

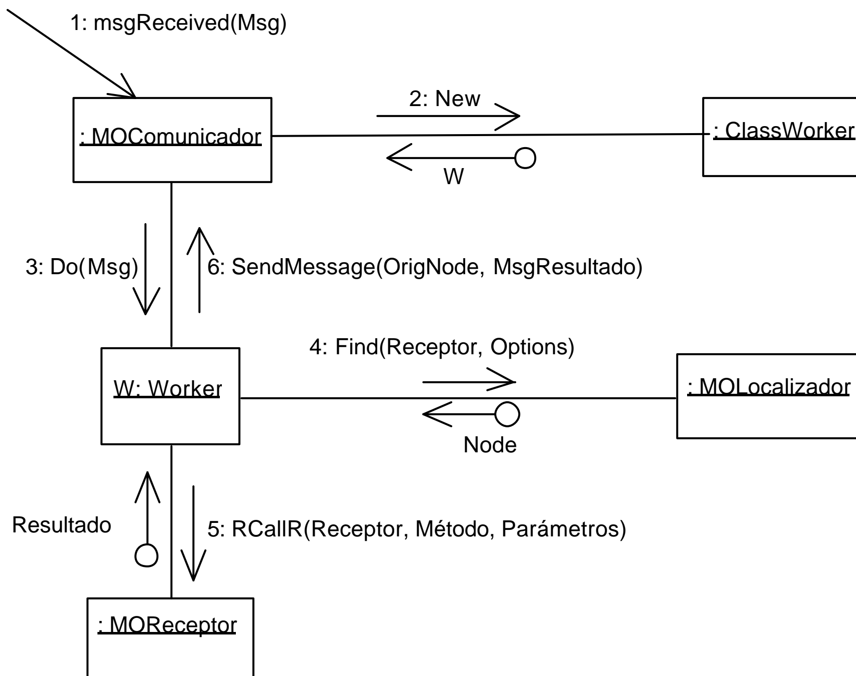


Figura 16.18. Parte remota de una invocación remota.

En el extremo receptor, otro meta-objeto comunicador es advertido de la llegada de un nuevo mensaje (paso 1). El meta-objeto comunicador crea un objeto trabajador (paso 2), encargado de una parte importante de la invocación en el nodo remoto (3).

El objeto trabajador se asegura de que el objeto a invocar se encuentra en el nodo invocando al meta-objeto localizador local (paso 4), dado que podría haber migrado antes de que llegara el mensaje de la invocación. En el caso de que se encuentre en el nodo, únicamente tiene que ceder el control al meta-objeto receptor del objeto servidor (paso 5), con lo que nos encontramos en la parte ya descrita de una invocación local de un método de usuario. Los parámetros utilizados en las invocaciones de los pasos 4 y 5 son obtenidos del mensaje obtenido por el meta-objeto comunicador en el paso 1.

Cuando finaliza la invocación del método en el objeto servidor, es preciso devolver el resultado al nodo origen. Para ello, se crea un mensaje de retorno con el resultado de la invocación que es entregado al meta-objeto comunicador local (6) para que se lo haga llegar al meta-objeto comunicador de la máquina en la que reside el objeto cliente. En la máquina origen, dicho resultado será reclamado desde el meta-objeto emisor al meta-objeto comunicador para hacérselo llegar, definitivamente, al objeto cliente.

CAPÍTULO 17 TRABAJO RELACIONADO

17.1 Introducción

En el presente capítulo se describen dos trabajos que guardan cierta relación con el desarrollado en esta Tesis, en algunos aspectos de la misma.

En primer lugar, el sistema Meta-java, que extiende la máquina virtual de Java con el fin de proporcionar reflectividad estructural y de comportamiento. La herramienta básica para su construcción es el modelo de eventos de Java.

En segundo lugar, se presenta la tecnología de agentes móviles, que está alcanzando un gran auge en los últimos años. Su relación con la distribución de objetos es muy importante, aunque siguen un camino claramente diferente al presentado en esta Tesis. Su base fundamental es mover únicamente el código ejecutable allá donde se encuentren los datos a procesar.

17.2 Meta-java

MetaJava[22] [Gol97] es un sistema desarrollado sobre la máquina virtual de Java (JVM, *Java Virtual Machine*) [Sun97a], que extiende el intérprete de Java permitiendo, no sólo reflectividad estructural proporcionada por su API reflectiva, sino también reflectividad del comportamiento.

17.2.1 Guías de diseño

El principio de diseño fundamental en MetaJava es que **el meta-sistema no debe efectuar acciones triviales, sino excepcionales**. De ello se derivan algunos criterios de diseño como el rendimiento, la generalidad de la arquitectura y la separación de los aspectos funcionales y los no funcionales.

- Rendimiento. Si no se utiliza, el meta-sistema no debe imponer ninguna penalización de rendimiento.
- Permitir en lo posible, la separación de los aspectos funcionales, específicos de la aplicación, de los no funcionales, como persistencia o replicación. Esta separación debe ser visible desde la fase de diseño a la de codificación. Tanto los programas del nivel base como los programas del meta-nivel debe ser reusables.
- La arquitectura debe ser general, es decir, problemas como distribución, sincronización y otros deben tener cabida y solución en ella.

17.2.2 Modelo computacional de MetaJava

Los sistemas tradicionales consisten en un sistema operativo y, sobre él, un programa que explota los servicios del mismo usando una API.

[22] Ahora denominado MetaXava por problemas de marcas comerciales.

La aproximación reflectiva propuesta por MetaJava es diferente. El sistema consiste en el sistema operativo, el programa de aplicación (sistema base) y el meta-sistema.

17.2.2.1 Separación en nivel base y meta-nivel

El programa está dividido en sistema base y meta-sistema, cada uno con un ámbito claramente separado.

- **Sistema base**. El programa de aplicación o sistema base no debe preocuparse del meta-sistema, sino dedicarse al problema de la aplicación.
- **Meta-sistema**. El meta-sistema se encarga de las computaciones que no solucionan directamente el problema de la aplicación, sino que solucionan los problemas de la computación en el nivel base.

17.2.2.2 Transferencia de control

Reflectividad implícita

Para comunicar el nivel base y el meta-nivel, MetaJava utiliza la emisión de eventos síncronos, es decir, la computación en el nivel base eleva eventos (del modelo de eventos de Java) que se envían al meta-sistema.

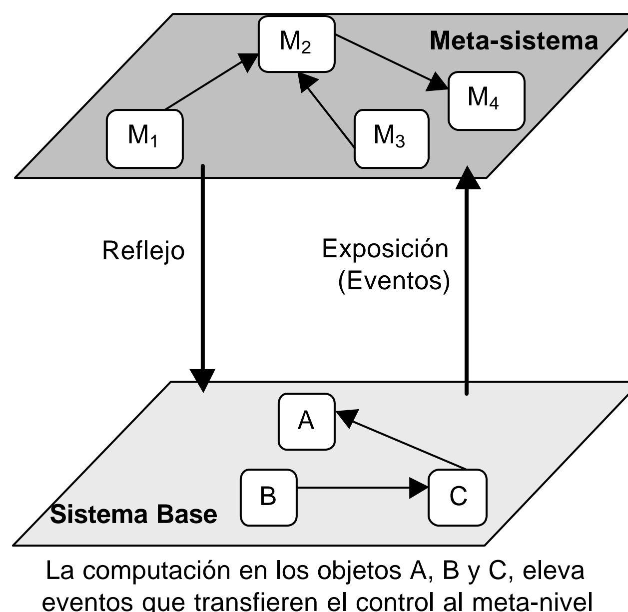


Figura 17.1. Modelo computacional de reflectividad del comportamiento.

El meta-sistema evalúa los eventos y reacciona de manera específica a ellos. Todos los eventos se gestionan de forma síncrona y la computación del nivel base se suspende mientras el meta-objeto procesa el evento. Esto da al meta-nivel control absoluto sobre la actividad del nivel base.

Por ejemplo, si el meta-nivel recibe el evento de que se ha invocado un método (`enter-method`), el comportamiento por omisión será ejecutar el método. Sin embargo, el meta-objeto podría también sincronizar la ejecución del método con otro método del objeto base. Otras alternativas serían encolar el método para retrasar su ejecución y retornar al llamador inmediatamente o ejecutar el método en otra máquina. Lo que suceda depende, completamente, del meta-objeto utilizado.

Reflectividad explícita

Un objeto base también puede invocar directamente un método en el meta-objeto. Es lo que se conoce como **meta-interacción explícita** y se utiliza para controlar el meta-nivel desde el nivel base.

La interfaz del meta-nivel de JVM define una serie de métodos agrupados según su función y que permiten el acceso a características del meta-nivel mediante la invocación del mismo. Se proporciona una lista resumida de los mismos más adelante.

17.2.3 Asociación de meta-objetos con las entidades del nivel base

Los meta-objetos pueden asociarse con objetos, clases y referencias. La asociación del meta-objeto con una entidad del nivel base significa que todos los eventos involucrados en la ejecución de esa entidad base son enviados al meta-objeto asociado. Si el meta-objeto está asociado a una clase, todos los eventos generados por cualquier instancia de esa clase son enviados al meta-objeto. Si está asociado a una referencia, solo se envían al meta-objeto los eventos generados por operaciones que utilizan esta referencia. La asociación es explícita, mediante las operaciones `attachObject`, `attachReference` y `attachClass`.

No todo objeto debe tener un meta-objeto asociado a él. Los meta-objetos pueden asociarse dinámicamente con los objetos del nivel base en tiempo de ejecución, lo que significa que las aplicaciones solo sufren la sobrecarga del meta-sistema si lo necesitan.

17.2.4 Estructura

La extensión MetaJava consiste en dos capas. La capa inferior abre la JVM y recibe el nombre de **Interfaz del Meta-nivel**, (del inglés, *meta-level interface*). La capa superior está escrita en Java y proporciona el mecanismo de eventos descrito, utilizando generación de código en tiempo de ejecución.

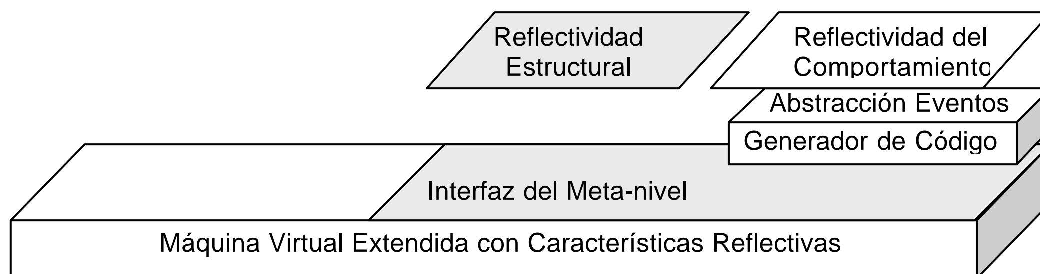


Figura 17.2. Estructura de MetaJava.

17.2.4.1 Interfaz del meta-nivel o MLI

Las principales funciones de la MLI se agrupan en los siguientes apartados.

- **Manipulación de datos del objeto**. El propósito es modificar el estado de un objeto. Se trata de funciones para acceder a las variables instancia de objetos y para acceder a los nombres y tipos de las mismas.
- **Soporte para modificación del código**. Se trata de funciones para recuperar los *bytecodes* de un método y modificarlos.

- **Enlace nivel base - meta-nivel**. Interfaz de bajo nivel para el enlace entre el nivel base y el meta-nivel que los meta-objetos usan para implementar la asignación.
- **Modificación de la estructura de clase**. Los meta-objetos necesitan cambiar la semántica de objetos o referencias, para lo que modifican las estructuras de la clase.
- **Modificación del conjunto de constantes**. Debe cambiar si cambian los métodos de una clase.
- **Ejecución**. Este grupo contiene funciones que posibilitan la ejecución de métodos arbitrarios.
- **Creación de instancias**. Se trata de métodos que registran la creación de un objeto.
- **Bloqueo de un objeto**. Se trata de métodos que bloquean y desbloquean una cerradura en un objeto.
- **Carga de clases**. Se trata de métodos que registran la carga de una clase, cargan la clase e instalan una nueva clase.

17.2.5 Reflectividad del comportamiento

La reflectividad del comportamiento se consigue asociando meta-objetos con los objetos del nivel base y transfiriéndoles el control ante determinados eventos.

Por tanto, la implantación de la reflectividad del comportamiento implica dos mecanismos: establecer la relación entre objetos y meta-objetos e implantar un sistema de eventos.

17.2.6 Implantación de la relación objetos base – meta-objetos: clases sombra

Para cambiar la semántica de un objeto sin afectar la semántica de otros objetos de la misma clase, el sistema MetaJava utiliza el concepto de clase como contenedor de toda la información acerca de un objeto, incluida la información relativa a su comportamiento. Para ello implanta la abstracción denominada **clase sombra** o ***shadow class***.

Una clase sombra C' es una copia exacta de la clase C con las siguientes propiedades:

- C y C' son indistinguibles desde el nivel base
- C' es idéntica a C excepto por modificaciones hechas por un meta-programa
- Los campos y métodos estáticos se comparten entre C y C'.

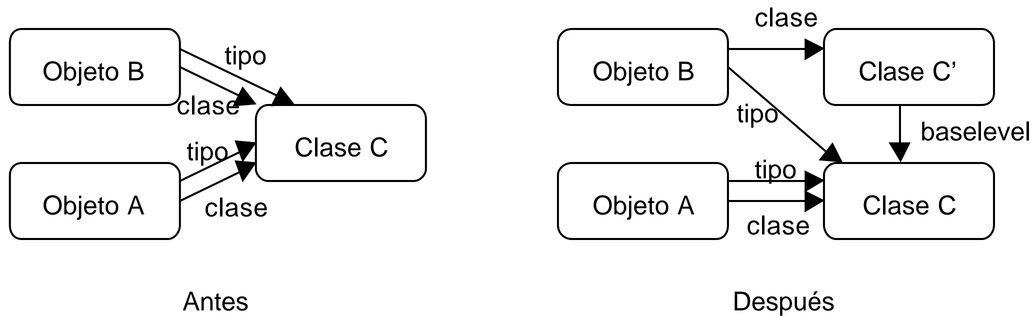


Figura 17.3. Creación de una clase sombra

Los lenguajes basados en clases suponen, normalmente, que existen muchos objetos del mismo tipo (clase). Los lenguajes basados en prototipos como Self [Cha92] o Moostrap soportan una especificación de reusabilidad distinta. Definen objetos *one-of-a-kind* [MMC95], donde es fácil derivar un objeto de otro y cambiar los campos o métodos del nuevo objeto, sin afectar al original. Las clases sombra de MetaJava son una aproximación a este comportamiento para un entorno basado en clases en lugar de en prototipos.

17.2.6.1 Asociación de meta-objetos a los objetos del nivel base

La Meta Java Virtual Machine (MJVM), almacena los objetos a través de manejadores de objetos (del inglés *handles*), que contienen punteros a los datos del objeto y a la clase del objeto.

Cuando se asocia un meta-objeto con un objeto de nivel base, se crea una clase sombra y el enlace del objeto a su clase se redirecciona a la clase sombra, a la vez que se asocia el meta-objeto con la clase sombra mediante el enlace `metaobject`.

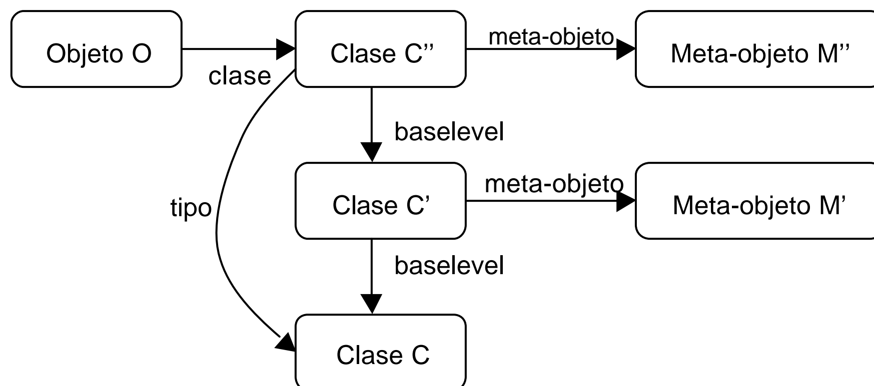


Figura 17.4. Enlace de objeto y meta-objeto: torre de meta-objetos.

17.2.6.2 Torre reflectiva

En MJVM, cada clase establece enlaces con el meta-objeto que tenga asociado, si tiene alguno, y con la clase base, si existe.

- **Enlace de tipo**. Cuando se crea un objeto, el enlace de tipo se establece automáticamente a la clase original, mientras el enlace de clase se establece a la clase correspondiente, C o C'.

- **Enlace `baselevel`**. Cada clase sombra tiene un enlace a su clase base.

La torre reflectiva en MJVM se construye siguiendo el enlace baselevel, que implanta una torre de meta-objetos como se puede apreciar en la figura anterior.

17.2.6.3 Problemas de la implantación

La utilización de clases sombra introduce algunos problemas:

- **Mantenimiento de la consistencia entre C y C'**. Todos los datos no constantes relativos a la clase deben compartirse entre C y C', dado que las clases sombra no deberían cambiar las clases sino propiedades de los objetos.
- **Recolección de clases sombra** cuando se desasocian del objeto.
- Algunos hilos pueden ejecutarse en un objeto base de clase C cuando se está creando y modificando la clase sombra. El sistema debe garantizar que el código original se mantiene y usa al menos, mientras se ejecute el código.

17.2.7 Transferencia del nivel base al meta-nivel: eventos

El sistema de generación de eventos es una capa sobre el MLI que proporciona la abstracción de reflexión del comportamiento.

La capa de generación de eventos para eventos locales se implementa completamente en términos de las facilidades de reflectividad estructural.

Si el meta-objeto está interesado en un comportamiento específico del objeto del nivel base, registra el evento con uno de los métodos definidos al efecto.

Eventos reflectivos

Actualmente, MetaJava define eventos para la llamada de métodos, para el acceso a campos en un objeto, para el bloqueo de objetos, para la carga de clases y para la creación de objetos.

17.2.8 Invocación de métodos remotos

La invocación de métodos remotos está implementada por la clase `MetaRemote` y una librería de clases que gestionan la comunicación remota. La clase `MetaRemote` es una implementación simple de la invocación remota.

En el caso de que un nodo exporte referencias a objetos, debe instalar un **object-server**. Un object-server gestiona las correspondencias existentes entre identificadores (ID) de objetos y objetos locales. Todo nodo que desee invocar un método de un objeto remoto debe instalar un object-server para los argumentos del método, ya que los argumentos se pasan por referencia.

En la figura siguiente se muestra qué ocurre cuando un objeto invoca un método en un objeto remoto. Antes de la invocación, el objeto que invoca habrá obtenido una referencia al objeto remoto en la forma de **representante** (*proxy*). El representante no es más que un objeto vacío, con un meta-objeto asociado y se utiliza únicamente para pasar el control al meta-nivel, donde está implementado el protocolo de comunicaciones.

El objeto `X` invoca el método `m1` del objeto remoto `O` (**`o.m1(A)`**). Esta acción, realizada en el nivel base, es implementada en el meta-nivel como sigue. Dado que el objeto `O` reside en un nodo distinto, `X` realmente invoca a O_{proxy} ①. El meta-objeto M_1 recibe el control, al estar asociado a O_{proxy} ②. M_1 registra el objeto argumento `A` en el

object-server S_1 y obtiene un manejador independiente de la localición para él ③. A continuación, invoca al object-server S_2 con dicho manejador, el nombre del método y el manejador para el objeto base `O` ④. El object-server S_2 instala el representante A_{proxy} para el objeto argumento en el nodo 2 y le asocia el meta-objeto M_2 ⑤ y ⑥. El object-server S_2 invoca al objeto del nivel base `O` ⑦. El objeto `O` procede a ejecutar el código del método y se encuentra con una llamada al método `m2` del objeto argumento `A` (**`A.m2(...)`**). La invocación es dirigida al representante A_{proxy} ❶, que produce una cesión de control al meta-objeto M_2 ❷ que envía la petición a S_1 en el nodo 1 ❸. S_1 busca el objeto local correspondiente a la referencia que recibe (`A`) y le cede el control, pasándose de nuevo al nivel base ❹.

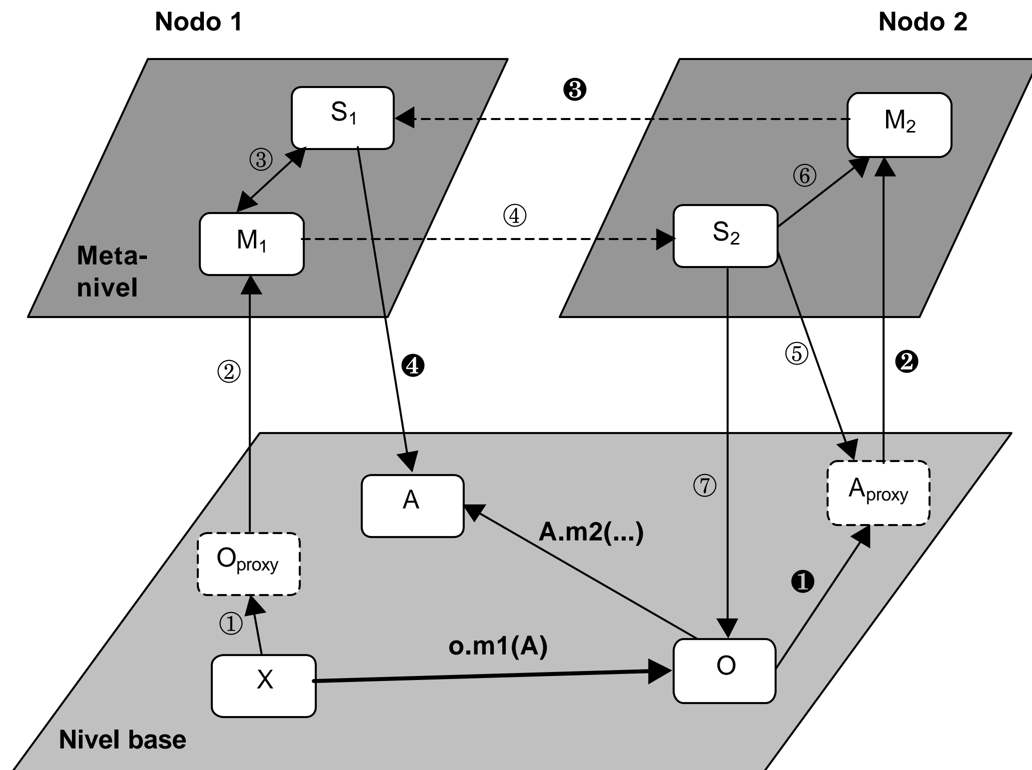


Figura 17.5. Invocación de métodos remotos en MetaJava.

17.2.9 Implantación

La implantación suele llevarse a cabo reemplazando los *bytecodes* con código intermedio (del inglés, *stub*) que salta al meta-espacio. Para ello, cuando el meta-objeto se registra para recibir un evento (por ejemplo, la invocación de un método en un objeto por parte del objeto actual), el generador de código busca los *bytecodes* correspondientes de las aplicaciones y los rescribe para que puedan beneficiarse de la reflectividad. En el ejemplo se ocuparía de crear el objeto evento e invocar a la función para despachar el evento.

De esta forma, cuando se produce el evento, el objeto salta al meta-nivel e invoca el método de la clase `MetaObject` para gestionar el evento.

17.3 Agentes móviles

17.3.1 Definición de agente

La primera definición de agente es la dada por G.W. Lecky-Thompson: "*Un agente es una pieza de software que ejecuta una tarea dada usando información obtenida de su entorno para actuar de forma apropiada para completar la tarea con éxito. El software debe ser capaz de adaptarse a sí mismo en base a los cambios que ocurren en su entorno, para que un cambio en circunstancias le permita aún obtener el resultado deseado*".

Un segundo intento más técnico orientado a la tecnología de objetos y agentes móviles es el dado por Fritz Hohl [HBS96]: "*Un agente móvil es un objeto especial que tiene un estado de datos (otros objetos no agentes, estructuras y bases de datos), un estado de código (las clases del agente y otras referencias a objetos) y un estado de ejecución (el control de procesos que se ejecutan en el agente)*"

Como ninguna de estas definiciones resulta completa, en vez de dar una definición formal, se suele proporcionar la lista de características que se espera que un agente deba tener, para poder tener una idea de lo que un agente puede ser. Las características siguientes suelen tenerlas los agentes móviles que serán el tipo de agentes a los que está dedicado este apartado.

- Autonomía: un agente opera [WJ95] sin la intervención directa de humanos y debe tener una cierta clase de control sobre sus acciones y su estado interno.
- Habilidad social: los agentes interaccionan con otros agentes y (posiblemente) con humanos.
- Reactividad: los agentes perciben su entorno y responden en un tiempo razonable a los cambios que ocurren en él. El agente puede estar en estado pasivo la mayor parte del tiempo y despertar al momento de que detecte ciertos cambios[23].
- Proactividad: los agentes no sólo responden a cambios sino que pueden tener un comportamiento con una iniciativa propia hacia una meta dirigida.
- Continuidad temporal: los agentes están constantemente ejecutando procesos ya sea en forma activa o pasiva.
- Orientación hacia el objetivo final: el agente es capaz de desarrollar una tarea compleja. Para lograrla es necesario subdividir esta tarea en pequeñas subtareas, el agente debe decidir por sí mismo la mejor manera y orden de ejecutarlas para lograr el objetivo final.
- Movilidad[24]: el agente debe ser capaz de suspender su ejecución en un servidor y reanudarla en otro servidor una vez que se haya desplazado a este. Este concepto se ha introducido en los últimos años.

Aunque aún no hay un solo agente que posea todas estas habilidades, existen sistemas de agentes prototipo que poseen muchas de ellas. Además nos proporcionan

[23] Esta característica es muy similar a la que tienen los demonios en el entorno UNIX

[24] Esta propiedad inicialmente se adjudicaba a los agentes en el ámbito de la IA. Sin embargo, en los últimos años ha sido el factor principal para el desarrollo de los agentes en el ámbito de Internet y ha llegado a ser considerada una característica fundamental

una buena idea de lo que un agente puede hacer, así como que características debemos de evaluar al examinar la calidad de un sistema de agentes en específico.

17.3.2 Movilidad

La movilidad es la facultad que tiene un agente para suspender su ejecución momentáneamente, trasladarse a otro servidor, volver a activarse y continuar su ejecución justo en donde la había detenido anteriormente. Al trasladarse debe llevar consigo sus datos, código y estado de ejecución para poder continuar la tarea que estaba desempeñando previamente. De igual forma es obvio suponer que todas las acciones para conseguir su traslado deben ser ejecutadas sin intervención humana.

Resulta oportuno mencionar que hoy en día los sistemas de agentes móviles más populares son los basados en Java[25]. Estos sistemas son capaces de transportar el estado de datos, es decir sus bases de datos y estructuras; de igual forma pueden transportar su estado de código, como lo son las clases y objetos que requiere, pero no su estado de ejecución, que se refiere al punto exacto que se estaba ejecutando al detener la ejecución del agente. Esto es debido a que la maquina virtual de Java no permite acceder a estos recursos.

Los **agentes móviles** son, por tanto, aquellos que ejecutan las tareas designadas en distintos ordenadores moviéndose de uno a otro para recopilar la información requerida y posteriormente regresar a su sitio de origen. Las características básicas de un agente móvil son:

- **Autonomía**: tienen su propio hilo y actúan por sí mismos.
- **Móvil***:* serializable.
- **Concurrente***:* puede ejecutarse con mas agentes a la vez.
- **Direccionable**: se puede definir su comportamiento.
- **Continuo**: se ejecutan continuamente y por tiempo indefinido.
- **Reactivo**: reacciona a su entorno mediante métodos.
- **Social**: interoperan con objetos y otros agentes.
- **Adaptativo***:* lo gestiona con excepciones.

17.3.3 El paradigma de agentes móviles

El principio de organización central de las comunicaciones en las redes de ordenadores de hoy en día son las **llamadas procedimientos remotos** o RPC (*Remote Procedure Call*), los cuales habilitan a un ordenador llamar procedimientos en otro (ver figura siguiente).

[25] Entre los que podemos citar Aglets Workbench, Voyager, Mole, Java-to-go, etc.

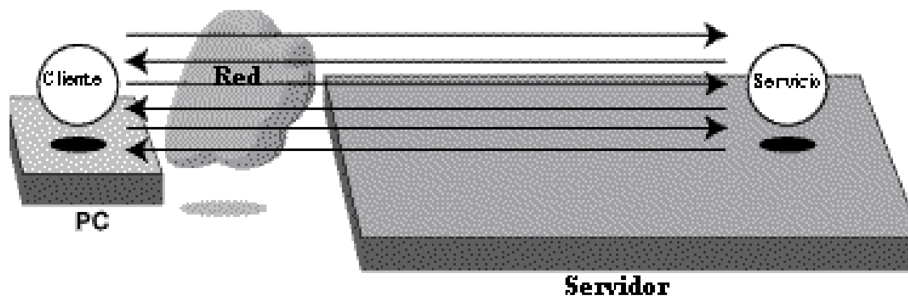


Figura 17.6. Aproximación mediante el uso de RPC

Cada mensaje en la red transporta solicitudes o confirmaciones de procedimientos. Una solicitud incluye datos que son los argumentos del procedimiento y la respuesta incluye datos que son sus resultados. El procedimiento mismo es interno al ordenador que lo ejecuta. Y todos los mensajes viajan desde el punto origen al destino

Una alternativa a las llamadas de procedimientos remotos es la promoción remota (obsérvese la figura siguiente). Dos servidores que se comunican con el paradigma de programación remota hacen un acuerdo sobre las instrucciones que son permitidas en un procedimiento y los tipos de datos que son permitidos en su estado. Estos acuerdos constituyen un lenguaje. El lenguaje incluye instrucciones que permiten al procedimiento tomar decisiones, examinar, y modificar su estado, y llamar procedimientos proporcionados por el ordenador que se está recibiendo el programa. Tales llamadas a procedimientos son locales en vez de remotas. El procedimiento y su estado son llamados **agente móvil** para enfatizar que ellos representan al ordenador que los envía aunque ellos residen y operan en el ordenador que les recibe.

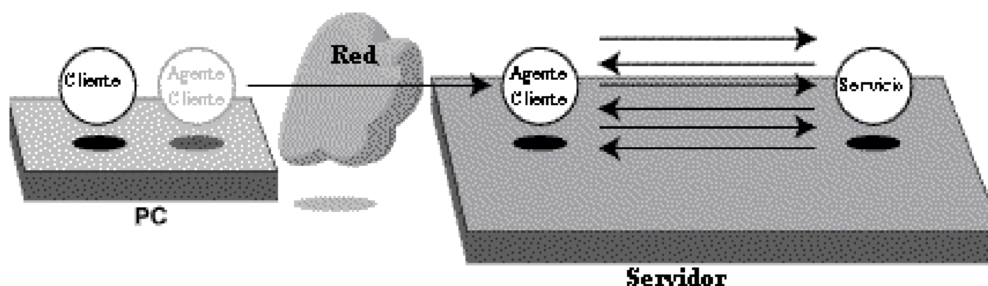


Figura 17.7. Nueva aproximación mediante programación móvil.

17.3.4 Conceptos de agentes móviles

La primera implementación comercial del concepto de agentes móviles fue la tecnología de Telescript de General Magic que intentó permitir acceso automático e interactivo a las redes de ordenadores usando agentes móviles [TV96].

Telescript fundó las bases de la tecnología de agentes móviles y es reconocido precursor de la misma. Estos conceptos fueron tan bien diseñados que han sido utilizados por los nuevos sistemas de agentes de nuestros días.

17.3.4.1 Lugares

La tecnología de agentes modela una red de ordenadores, tan grande como una colección de lugares que ofrecen un servicio a los agentes móviles que entran en él. Así pues un lugar es un sitio de interacción de agentes que les ofrece un servicio y les

facilita la consecución de sus atareas a todos aquellos agentes a los que les haya sido permitido entrar. En la tecnología de telescript un lugar era ocupado permanentemente por un agente distinguido. Este agente estacionario representaba al lugar y proveía su servicio (véase la Figura 17.8).

17.3.4.2 Agentes

Cada agente móvil que fue definido anteriormente, ocupa un lugar particular. Sin embargo un agente puede moverse de un lugar a otro. De esta forma ocupa lugares diferentes en tiempos diferentes. Los agentes son independientes por lo que sus procedimientos son ejecutados concurrentemente.

Figura 17.8. Representación de lugares y agentes.

17.3.4.3 Viajes

A los agentes les es permitido viajar de un lugar a otro, sin importar la distancia. Esto es el sello distintivo de un sistema de programación remota. De esta forma un viaje le permite a un agente obtener un servicio ofrecido remotamente y regresar a su lugar de origen.

17.3.4.4 Entrevistas (meetings)

A dos agentes les es permitido entrevistarse si se encuentran en el mismo lugar. Una entrevista permite a los agentes en el mismo ordenador llamar a los procedimientos de otros.

Figura 17.9. Viajes y entrevistas

Las entrevistas motivan a los agentes a viajar. Un agente podría viajar a un lugar en un servidor para entrevistarse con el agente estacionario que provee el servicio que el lugar ofrece. La figura anterior describe las interacciones en los viajes y entrevistas.

17.3.4.5 Conexiones

Una conexión de comunicación entre dos agentes en lugares distintos, es con frecuencia hecha en beneficio de los usuarios humanos para que interactúen con las aplicaciones.

La instrucción de conexión permite a los agentes de los usuarios el intercambio de información a distancia. Por ejemplo un agente podría enviar información al sitio de donde proviene (a través de otro agente) para que el usuario origen pueda seleccionar alguna opción de la información que ha encontrado.

17.3.4.6 Autoridades

La autoridad de un agente o lugar en el mundo electrónico es el individuo, organización, entidad o empresa a quien representa dentro del mundo físico. Una autoridad puede ser propietaria de varios agentes. La autenticación de los agentes generalmente consiste en descubrir su autoridad. Los agentes y los lugares pueden discernir, pero nunca podrán negar ni falsificar sus autoridades, evitando el anonimato.

17.3.4.7 Permisos

Las autoridades pueden limitar los que los agentes y lugares puede hacer asignándoles permisos a ellos. Un permiso es un dato que acepta capacidades. Un agente o lugar puede darse cuenta de sus capacidades, lo que le es permitido hacer, pero nunca podrá incrementarlas.

17.3.5 Tecnología de agentes móviles

Una tecnología para agentes móviles es una **infraestructura de software** que puede montarse encima de una gran variedad de computadores y hardware de comunicaciones, presente y futuro. Esta tecnología, implementa los conceptos mencionados en la sección anterior y otros relacionados con ellos que les permite a los agentes interoperar.

La tecnología tiene tres componentes principales: el **lenguaje** en el cual los agentes y los lugares son programados; una **máquina** o intérprete para ese lenguaje; y los **protocolos de comunicación** que permiten a esa máquinas residir en diferentes ordenadores para lograr el envío en intercambio de agentes.

17.3.5.1 El lenguaje

El lenguaje de programación de los creadores de aplicaciones de comunicación con agentes debe definir los algoritmos que los siguen los agentes y la información que llevan conforme viajan por la red.

Para facilitar el desarrollo de las aplicaciones de comunicación, y la interacción entre lugares y agentes el lenguaje de programación debe ser:

- **Completo***:* para que cualquier algoritmo pueda ser expresado en el lenguaje.
- **Orientado a objetos***:* para obtener los beneficios de esta tecnología
- **Dinámico***:* para que pueda transportar la clases que se requieran para crear instancias de agentes en máquinas remotas.
- **Persistente***:* para que el agente y su información sean respaldados en un medio no volátil.
- **Portable y seguro**: para que se pueda ejecutar sobre cualquier plataforma de una forma segura.

- **Versátil**: que permita tareas complejas de transporte, autenticación y control de acceso.

17.3.5.2 La máquina o intérprete

La máquina es un programa de software que implementa el lenguaje para mantener y ejecutar los lugares dentro de su contexto, al igual que a los agentes que ocupan esos lugares.

Al menos conceptualmente, la máquina consta de tres interfaces de programación de aplicaciones (API, *Application Programming Interface*). Obsérvese la Figura 17.10. Una API de almacenamiento que le da acceso a memoria no volátil que requiere para preservar a los lugares y agentes en caso de fallos. Una API de transporte que le da a la máquina acceso a los medios de comunicación que requiere para transportar agentes a y de otras máquinas. Y una API de aplicaciones externas que le permite a las partes escritas en el lenguaje de agentes interactuar con otras que no están escritas en dicho lenguaje.

17.3.5.3 Protocolos

Los protocolos habilitan a dos máquinas que se comuniquen. Las máquinas se comunican para transportar agentes. Los protocolos deben operar sobre una gran cantidad de redes de transporte (en la actualidad, sobresalen entre ellas las basadas en el protocolo de Internet TCP/IP).

Los protocolos operan en dos niveles. El nivel inferior gestiona el transporte de los agentes y el más alto, su codificación y decodificación. Esto se aprecia en la siguiente figura:

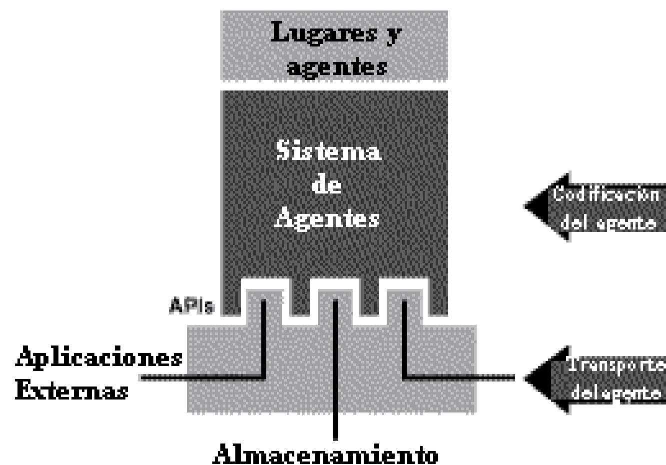


Figura 17.10. La máquina y los niveles de los protocolos

El protocolo de interconexión de plataformas especifica cómo dos máquinas se autentifican una a otra usando por ejemplo, criptografía de clave pública y entonces transfiere la codificación de un agente de una a otra.

17.3.6 Particularidades de los agentes móviles para la construcción de sistemas distribuidos

Los agentes móviles proporcionan un enfoque claramente diferenciado del tradicional para la construcción de sistemas distribuidos. A continuación se describen aquellos aspectos claramente particulares de este nuevo enfoque:

- Los agentes se ejecutan de forma asíncrona y autónoma: frecuentemente los dispositivos móviles tienen que depender de conexiones de red caras o frágiles. Esto es, tareas que requieren una conexión abierta continuamente entre un dispositivo móvil y una red no serán en la mayoría de los casos económica o técnicamente factibles. Las tareas pueden ser incrustadas en agentes móviles, que serán despachados hacia la red. Después de ser enviados, los agentes móviles llegan a ser independientes de la creación de procesos y pueden operar de forma asíncrona y autónoma. El dispositivo móvil puede conectarse nuevamente más tarde para consultar al agente el resultado de la tarea.
- Se adaptan dinámicamente. Los agentes móviles tienen la habilidad de percibir su entorno y reaccionar de forma autónoma a cambios. Un conjunto de agentes móviles posee la habilidad única de distribuirse a sí mismos entre los servidores en una red de tal forma que mantienen la configuración óptima para la solución de un problema particular.
- Son heterogéneos por naturaleza. La computación en redes es fundamentalmente heterogénea, con frecuencia desde las perspectivas de hardware y software. Como los agentes móviles son generalmente independientes del ordenador y de la capa de transporte y dependientes sólo de su entorno de ejecución, proporcionan las condiciones óptimas para una integración transparente.
- Son robustos y tolerantes a fallos. La habilidad de los agentes móviles para reaccionar dinámicamente a situaciones desfavorables y eventos hace más fácil construir sistemas distribuidos robustos y tolerantes a fallos. Si un servidor está siendo apagado, todos los agentes que se ejecutan en ese servidor serán avisados y se les dará tiempo suficiente para que se envíen a sí mismos y continúen sus operaciones en otro servidor de la red.

17.3.7 Aplicaciones de los agentes móviles

Los agentes móviles pueden ser utilizados para desarrollar una gran cantidad de aplicaciones. Estas aplicaciones pueden estar basadas en la tecnología de agentes, o bien pueden ser complementos de aplicaciones basadas en las tecnologías de orientación a objetos. Es decir, puede haber un sistema convencional que para realizar búsquedas remotas utilice la tecnología de agentes móviles.

De una u otra forma las aplicaciones más importantes que se pueden llevar a cabo con agentes móviles están entre las siguientes [Ven97]:

- Recolección de datos de distintos sitios. Una de las mayores diferencias entre el código móvil, como los *applets* de Java, y los agentes móviles es el itinerario. Mientras que el código móvil usualmente viaja sólo de un punto a otro, los agentes móviles tienen un itinerario y pueden viajar secuencialmente a muchos servidores. De ahí que una aplicación natural de los agentes móviles sea la recolección de información a través de muchos computadores enlazados a una red. Un ejemplo de esta clase de aplicación es una herramienta de copias de seguridad que periódicamente debe supervisar cada disco instalado en cada

computador que se encuentra enlazado a la red. Aquí, un agente móvil podría navegar la red, recolectar la información acerca del estado de la copia de seguridad de cada disco y entonces regresar a su posición de origen y hacer un informe.

- Búsqueda y filtrado. Dado el constante incremento en la cantidad de información disponible en Internet y en otras redes, la actividad de recolectar información de una red implica la búsqueda entre grandes cantidades datos de unas cuantas piezas relevantes de información. Eliminar la información irrelevante puede ser un proceso que consume mucho tiempo. En nombre de un usuario, un agente móvil puede visitar muchos servidores, buscar a través de la información disponible en cada servidor, y construir un índice de enlaces a las piezas de información que concuerdan con el criterio de búsqueda. El filtrado y la búsqueda muestran un atributo común a muchas aplicaciones potenciales de agentes móviles: el conocimiento de las preferencias del usuario.
- Monitorización. En algunas ocasiones la información no está distribuida a través de un espacio como puede ser un conjunto de ordenadores, sino a través del tiempo. Nueva información constantemente está siendo generada y publicada en la red. Los agentes pueden ser enviados para esperar por ciertas clases información hasta que ésta sea generada o se encuentre disponible. Por ejemplo un agente personalizado para la reunión de noticias. Un agente podría monitorizar varias fuentes de noticias para determinado tipo de información de interés para su usuario, e informarle cuando alguna información relevante esté disponible.
- Comercio electrónico. El comercio electrónico es otra aplicación apropiada para el uso de la tecnología de agentes móviles. Un agente móvil podría realizar compras, incluyendo la realización de órdenes de compra y potencialmente pagar. Por ejemplo, si se quiere volar de un sitio a otro, un agente podría visitar las bases datos de los horarios de vuelo y los precios de varias líneas aéreas, encontrar el mejor precio y horario de salida, hacer la reserva e incluso pagar con un número de tarjeta de crédito.
- Supercomputadora virtual (cálculos en multiprocesos). Los cálculos complejos con frecuencia pueden ser descompuestos en unidades discretas para la distribución en una pila de servidores o procesos. Cada una de estas unidades discretas puede ser asignada a un agente, el cual es entonces enviado a un sitio remoto en donde el trabajo es realizado. Una vez terminado, cada agente puede regresar a casa con los resultados, que pueden ser agregados y sumados.

CAPÍTULO 18 CONCLUSIONES

18.1 Introducción

Al inicio de este trabajo se identifican los Sistemas Distribuidos como herramientas para maximizar el rendimiento global de un conjunto de computadores geográficamente dispersos y unidos por algún tipo de red de comunicaciones. En la actualidad, la utilización de computadores y redes de comunicaciones es generalizada, pero no existen muchos sistemas que gestionen su complejidad y ofrezcan al usuario una interfaz sencilla para su utilización.

El primer problema a resolver se presenta a la hora de decidir cómo construir el sistema distribuido. Las tecnologías orientadas a objetos se presentan en la actualidad como una buena herramienta para el diseño y construcción de sistemas software de todo tipo. Muchos sistemas operativos y herramientas software se están construyendo en la actualidad haciendo uso de tecnologías orientadas a objetos, pero su aplicación es, en general, parcial.

En segundo lugar, y a pesar de que de alguna manera el sistema distribuido esté construido utilizando algún tipo de tecnología orientada a objetos, los servicios que aquel proporciona se siguen ofreciendo, en muchos casos, de la manera tradicional. Los usuarios de dichos sistemas operativos y herramientas se ven obligados a utilizar servicios proporcionados con algunas características de orientación a objetos y otras más clásicas.

Cabe preguntarse si es posible utilizar las tecnologías orientadas a objetos de una manera integral para la construcción de sistemas distribuidos. La respuesta está en los Sistemas Integrales Orientados a Objetos (SIOO), cuya arquitectura y funcionalidad se describen en este trabajo. Evidentemente, la distribución será una característica esencial de tales sistemas que se concretará en la distribución de sus componentes principales y únicos: los objetos.

La introducción de la distribución de los objetos en el sistema integral debe realizarse de tal manera que no afecte al funcionamiento de otros mecanismos. Además, se pretende que sea adaptable o configurable al entorno en el que se va a utilizar. La utilización de la reflectividad se ha mostrado como la solución para introducir de la forma descrita la distribución.

Quedan totalmente separados, desde este momento, el espacio (nivel) de objetos del usuario, en el que residen los objetos que describen las tareas de usuario o aplicaciones del meta-espacio (o meta-nivel), en el que residen los objetos que describen como se han de gestionar los objetos del espacio de usuario. A pesar de esta división, se sigue manteniendo la uniformidad conceptual en torno a la orientación a objetos, donde el objeto es la única entidad.

Algunos de los objetos del meta-espacio se responsabilizarán de las tareas derivadas de la distribución de objetos. Sus dos tareas básicas serán las de proporcionar capacidades de migración e invocación remota a los objetos del sistema integral de

manera transparente. Con la palabra transparencia se quiere resaltar el hecho de que los objetos no tengan porqué conocer el carácter distribuido del sistema integral para realizar su computación, de manera que puedan interaccionar como en un sistema centralizado convencional. Ahora bien, aquellos objetos que deseen sacar partido de la existencia de distribución en el sistema integral, tendrán los medios a su alcance para conseguirlo.

La existencia de otros mecanismos en el sistema integral, como la persistencia o la seguridad no deben ser un obstáculo para la distribución, pero es necesario estudiar como se integran.

Existen multitud de cuestiones relacionadas de alguna manera con la distribución en general y con la de objetos en particular. Es preciso estudiar de qué manera se pueden compatibilizar con el diseño presentado para la distribución.

Los capítulos 15 y 16 describen la implementación de un prototipo como el mostrado a lo largo del documento.

18.2 Resultados destacables

A continuación se resumen los resultados más destacables, agrupados en diferentes apartados.

18.2.1 Arquitectura reflectiva del sistema integral

El sistema integral se compone de un conjunto de máquinas abstractas y un conjunto de objetos que hacen el papel de sistema operativo. Las máquinas abstractas ofrecen la funcionalidad básica, aquella que resultaría cara, peligrosa o difícil de implantar en cualquier otra capa del sistema integral. El sistema operativo extiende el nivel básico, es decir, añade o modifica la funcionalidad codificada internamente en las máquinas, adaptando su comportamiento a un entorno concreto. La reflectividad se ha mostrado como el mecanismo más prometedor para obtener un entorno de computación flexible resultado de la unión de máquinas abstractas y sistema operativo.

Además, la utilización de la reflectividad permite obtener una arquitectura abierta, de tal manera que de manera sencilla se puede extender el conjunto de servicios que proporciona el sistema, sin duplicar ni interrumpir los ya existentes. Incluso es posible cambiar las políticas que siguen determinados mecanismos sin necesidad de cambiar el código de las aplicaciones.

18.2.2 Introducción reflectiva de la distribución

La distribución de objetos se presenta como una característica fundamental en los sistemas operativos orientados a objetos modernos. Un sistema integral oriental orientado a objetos, que presenta, entre otras cosas, la funcionalidad de un sistema operativo, debe proporcionar también la mencionada distribución.

En un sistema integral construido con una arquitectura reflectiva, la distribución de objetos se considera como una de las posibles extensiones del modelo de objetos básico y tradicional. Habrá de ser, por tanto, implementado al nivel del sistema operativo.

El sistema de distribución de objetos del sistema integral toma la forma de un conjunto de objetos que residen en el meta-nivel y que están asociados a objetos del nivel base.

18.2.3 Transparencia

La transparencia de todo tipo siempre se presenta como un objetivo a la hora de diseñar un sistema distribuido. Refiriéndose a un sistema integral orientado a objetos y distribuido, se busca la transparencia para todas aquellas operaciones que pueda realizar un objeto.

La transparencia en la invocación de objetos quiere decir que un objeto puede ser invocado por otro con independencia de sus respectivas ubicaciones. Dicha transparencia es conseguida por el sistema de distribución modificando el comportamiento que por defecto presentan las máquinas abstractas para la invocación. La modificación es realizada por objetos del sistema operativo que son los que se enfrentan a la complejidad distribuida del sistema integral, convirtiendo invocaciones supuestamente locales para un objeto en invocaciones realmente remotas. En este sentido, tanto el objeto que realiza la invocación como el que es invocado son totalmente ajenos a la naturaleza real de la invocación. Esto es muy importante para la mayor parte de los objetos, dado que liberan al programador de todas las dificultades que presenta la gestión de invocaciones remotas y puede concentrarse en la resolución propiamente dicha del problema.

La transparencia en la migración de objetos es conseguida al no ser conscientes estos de que realmente se está llevando a cabo (o ya se ha realizado). Todo objeto puede ser elegido en cualquier momento para ser migrado de una a otra máquina abstracta del sistema integral. Con independencia de la razón por la que se vaya a migrar un objeto, el objeto debe poder continuar con su computación en el nodo destino como si no hubiese sido interrumpido en ningún momento. De nuevo, la arquitectura reflectiva del sistema integral permite suspender la computación de un objeto, sacar una copia de la misma y enviarla al nodo destino de la migración, donde se reanudará como si nada hubiera ocurrido.

18.2.4 Control de la distribución para ciertas aplicaciones

A pesar de que la transparencia en la distribución es muy deseable, existen ciertas aplicaciones que desean sacar partido de su existencia. Es necesario, por tanto, que determinados aspectos de la distribución se hagan visibles para dichas aplicaciones.

En el caso del sistema integral, algunos objetos pueden desear ejercer cierto control sobre su ubicación particular o sobre la de algunos objetos con los que se relacionan. Se proporciona a estos objetos un medio por el que pueden solicitar la migración de objetos, solicitudes que pueden ser denegadas en el caso de que se concluya que serían contraproducentes para el sistema en el caso de ser llevadas a cabo. Además de especificar el nodo destino de la migración de los objetos, el sistema operativo puede proporcionar mecanismos de agrupamiento de objetos, con el fin de que los grupos migren siempre de manera solidaria una vez establecidos.

18.2.5 Uniformidad

Se mantiene totalmente la uniformidad en la orientación a objetos del sistema. La distinción de dos niveles, básico y meta-nivel, no rompe la uniformidad en la orientación a objetos, dado que todos siguen siendo objetos del modelo único. Simplemente unos proporcionarán funcionalidad de nivel usuario (las denominadas aplicaciones) y otros las que comúnmente se atribuyen al sistema operativo.

18.2.6 Adaptabilidad

Los mecanismos de invocación remota y migración de objetos que se ofrecen tienen una semántica mínima, con el fin de que puedan ser completados por lenguajes de programación o personalidades de sistemas operativos que necesiten proporcionar una semántica más compleja. Por ejemplo, para el caso de la invocación remota, el paso de parámetros se realiza siempre por dirección. Un paso de parámetros por valor se consigue con una operación de duplicación del objeto parámetro, y una operación de migración del mismo, si la invocación es remota. Implementar una única semántica en el paso de parámetros es más económico y la arquitectura del sistema permite igualmente que aquella sea adaptada a los requerimientos concretos de cada entorno.

18.2.7 Flexibilidad

Ninguno de los mecanismos presentados presupone unas políticas determinadas para su funcionamiento, aunque siempre van a necesitar de alguna. La deseada separación de mecanismos y políticas se traduce en un incremento de la flexibilidad, al ser posible reemplazar las políticas que rigen los mecanismos de forma no traumática. Las políticas serán implementadas como un conjunto de objetos, con una interfaz determinada para los objetos que implementan los mecanismos. Las políticas podrán ser cambiadas siempre y cuando los nuevos objetos respeten la interfaz proporcionada por los antiguos.

Además, dado que la funcionalidad del sistema operativo es proporcionada por un conjunto de objetos asociados a objetos del nivel base, es posible modificar características del sistema operativo para un objeto o un conjunto de ellos sin afectar al comportamiento que el sistema operativo sigue presentando para el resto. Esta posibilidad permitiría, por ejemplo, la coexistencia de diferentes personalidades de sistemas operativos que presentasen aspectos semánticos claramente diferentes.

18.3 Trabajo y líneas de investigación futuras

Esta Tesis sienta las bases del sistema de distribución de objetos de un sistema distribuido, pero deja abiertas diferentes líneas de investigación, para completar y mejorar lo ya existente y para desarrollar nuevos elementos. A continuación se reseñan algunas de las líneas más inmediatas relacionadas con los temas discutidos en esta Tesis.

18.3.1 Interoperabilidad con otros sistemas de soporte de objetos

El sistema distribuido de objetos inmerso en el sistema integral es un sistema cerrado, en el sentido de que los objetos sólo tienen la posibilidad de interaccionar con otros objetos del sistema integral.

Existen en el mercado dos sistemas de objetos distribuidos que cuentan con el respaldo de un gran número de empresas y programadores: DCOM, de Microsoft, y CORBA, de OMG. El objetivo principal de esto sistemas de soporte de objetos distribuidos es permitir obtener los beneficios de la orientación a objetos desde lenguajes de programación convencionales con el objetivo de construir aplicaciones distribuidas.

El trabajo en este aspecto consistiría en la construcción del software necesario (conjunto de objetos) en el sistema integral que permitiera su conexión con los sistemas de objetos mencionados, rompiendo de esta manera su aislamiento. Evidentemente, el

grado de dificultad de la tarea es grande, dado que se trata de cumplir unos estándares internacionales que, en el caso de CORBA, están en continua modificación y, en el caso de DCOM, son propietarios de Microsoft.

18.3.2 Construcción de compiladores de lenguajes

Es razonable pensar que los usuarios del sistema integral no construirán sus aplicaciones utilizando directamente el lenguaje ensamblador de la máquina abstracta, sino que utilizarán lenguajes de programación de alto nivel (orientados a objetos o no).

Partiendo de este hecho, se pueden realizar diferentes trabajos enfocados a que los compiladores y lenguajes diseñados saquen partido a la distribución.

18.3.2.1 Lenguajes con control de la distribución

Este tipo de lenguajes, que podrían crearse como la extensión de alguno de alguno de los ya existentes, posibilitan al programador controlar diferentes aspectos de la distribución de las entidades que maneja el lenguaje. El compilador se encargaría de convertir dichas sentencias de control en invocaciones a objetos del sistema operativo responsables de la distribución. En fácil pensar que la conversión se realizaría de una manera más sencilla si el lenguaje de programación fuera orientado a objetos.

18.3.2.2 Compiladores que hacen uso de la distribución

Es posible pensar también en la construcción de compiladores que sean conscientes de la existencia de la distribución en el sistema integral y que hagan uso de ella, aunque el lenguaje de programación al que dan soporte sea totalmente ajeno a la misma. Por ejemplo, en el lenguaje Pascal la semántica del paso de parámetros a funciones y procedimientos es doble: por valor y por referencia. El compilador de Pascal a construir puede sacar partido a su conocimiento de la existencia de la distribución, haciendo que las llamadas a procedimientos que se convierten en invocaciones a métodos remotos y que incluyen parámetros por valor migren las copias de los parámetros de manera automática, optimizando así dichas llamadas. De una forma similar se pueda hacer con los parámetros por referencia y con el resultado de la invocación.

18.3.3 Implementación de servicios

En el capítulo 13 se mencionan diferentes servicios relacionados con la distribución que de alguna manera son ofrecidos por algunos sistemas distribuidos (los más importantes). Una vez se complete de manera eficiente el sistema de distribución básico (Agra) del sistema integral, se puede afrontar la tarea de implantar dichos servicios en la forma apuntada.

El servicio de nombrado, aunque muy elemental, ya existe en los prototipos realizados. El objetivo en este caso sería acercar su funcionalidad a la de servicios de nombrado más populares como DNS, X.500, etc.

El servicio de replicación de objetos no presenta, en un principio, una excesiva complejidad. La implantación del servicio se realizaría, en una primera fase, para los objetos inmutables, que deberían ser identificados por el programador. En una segunda fase, se podrían contemplar el resto de los objetos, introduciendo protocolos de sincronización de réplicas.

El servicio de transacciones se presenta a priori como el más complejo. Para su implantación necesitaría, aparte de lo expresado en el capítulo 13, una eficiente

implementación del sistema de persistencia y, quizás, la de un sistema de gestión de bases de datos orientadas a objetos.

18.3.4 Introducción de otros servicios relacionados con la distribución

Las últimas versiones de los sistemas de soporte de objetos distribuidos comerciales más importanes (CORBA y COM+) vienen acompañadas de un conjunto, cada vez más importante, de servicios para la utilización e implementación de los objetos. Algunos de ellos ya están incluidos en el apartado anterior. Otros, como el servicio de tiempo (*Time Service*), el de relaciones (*Relationship Service*), el de colecciones (*Object Collections Service*), etc., podrían ser implantados también en el sistema operativo.

18.3.5 Desarrollo completo de un entorno de usuario

Siempre que se habla de un sistema operativo o de parte de él, es necesario hablar de la interfaz con el usuario. Se trata en este caso de posibilitar al usuario la utilización de los servicios del sistema integral (fundamentalmente, la distribución de objetos) sin necesidad de tener que escribir código, es decir, a través de una interfaz similar a las existentes en sistemas operativos tradicionales.

Con el fin de proporcionar al usuario la ilusión de estar trabajando en un entorno tradicional, tipo Unix o Windows, sería necesario también implementar políticas relacionadas con la distribución que permitiesen ofrecer dicha ilusión. De esta manera, se consigue además verificar el comportamiento del sistema ante la coexistencia de diferentes políticas para los mismos mecanismos aplicadas sobre diferentes conjuntos de objetos.

Referencias

[ABB+86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian y M. Young. Mach: A New Kernel Foundation for Unix Development. Summer USENIX Conference. Atlanta, 1986

[ABB+93] H. Assenmacher T. Breitbach, P. Buhler, V. Hübsch y R. Schwarz. "The PANDA Sytem Architecture: A Pico-Kernel Approach". En Proceedings of the *4th Workshop on Future Trends of Distributed Computing Systems*. IEEE Computer Society Press. Septiembre de 1993. Pág. 470-476.

[ADA+99] Darío Álvarez Gutiérrez, María Ángeles Díaz Fondón, Fernando Álvarez García y Lourdes Tajes Martínez. Eliminating Garbage Collection by Using Virtual Memory Tecniques to Implement Complete Object Persistence. 13th European Conference in Object Oriented Programming (ECOOP'99), Workshop on Object Orientation and Operating Systemas. Lisboa, Portugal. Junio de 1999

[AF89] Y. Arsty, R. Finkel. Designing a Process Migration Facility. The Charlotte Experience. Computer, Vol. 9, pp. 47-56, 1989.

[AK95] J.M.Assumpcao Jr y S.T.Kofuji. "Bootstrapping the Object Oriented Operating System Merlin: Just Add Reflection". En Proceedings of the European Conference on Object Oriented Programming (ECOOP'95), Meta'95 Workshop on Advances in Metaobject Protocols and Reflection. 1995

[Álv98] Darío Álvarez Gutiérrez. Persistencia completa para un sistema operativo orientado a objetos usando una máquina abstracta con arquitectura reflectiva. Tesis Doctoral. Departamento de Informática, Universidad de Oviedo, Marzo de 1998

[ANS89] ANSA. The Advanced Network Systems Architecture (ANSA) Reference Manual. Architecture Project Management. Castle Hill, Cambridge England. 1989.

[ATA+96] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Juan Manuel Cueva Lovelle y Raúl Izquierdo Castanedo."Un sistema operativo para el sistema orientado a objetos integral Oviedo3". Actas de las II Jornadas de Informática. Almuñécar, Granada. Julio de 1996

[ATA+97] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Raúl Izquierdo Castanedo y Juan Manuel Cueva Lovelle. "An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System". Eleventh European Conference in Object Oriented Programming (ECOOP'97), Workshop in Object Orientation in Operating Sytems. Jyväskylä, Finlandia. Junio de 1997.

[ATD+2000] Fernando Álvarez García, Lourdes Tajes Martínez, María Ángeles Díaz Fondón, Darío Álvarez Gutiérrez. Introducing Distribution in an Operating System Environment of Reflective Object Oriented Abstract Machines. 14th European Conference in Object Oriented Programming (ECOOP'2000), Workshop on Object Orientation and Operating Systemas. Niza, Francia. Junio de 2000

[ATD+98a] F.Álvarez García, L.Tajes Martínez, M.A.Díaz Fondón, D.Álvarez Gutiérrez y J.M.Cueva Lovelle. "Agra: The Object Distribution Subsystem of the SO4 Object-Oriented Operating System". En Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98). H.R.Arabnia, ed. pp. 255-258. CSREA Press. 1998

[ATD+98b] F.Álvarez García, L.Tajes Martínez, M.A.Díaz Fondón, D.Álvarez Gutiérrez y J.M.Cueva Lovelle. "Object Distribution Subsystem for an Integral Object-Oriented System". En Anales del Simposio Argentino en Orientación a Objetos(ASOO'98). Pp. 219-227. Buenos Aires, Argentina, Octubre de 1998

[BBD+91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme. Architecture and Implementation of Guide, an Object-Oriented Distributed System. Computing Systems, Vol. 4, nº 1, pp. 31-67, Invierno de 1991

[BC87] J-P.Briot y P.Cointe. “A Uniform Model for Object-Oriented Languages Using the Class Abstraction”. IJCAI’87. Agosto, 1987.

[BC89] J-P.Briot y P.Cointe. “Programming with Explicit Metaclasses in Smalltalk-80”.En Proceedings of the Object-Oriented Programming, Languages, Systems and Applications (OOPSLA’89). SIGPLAN Notices, 24:419-431, Octubre 1989

[BG96] Jean-Pierre Briot, Rachid Guerraoui. “A Classification of Various Approaches for Object-Based Parallel and Distributed Programming”. Techical Report. University of Tokyo and École Polytechnique Fédérale de Lausanne. 1996.

[BGW93] D.G.Bobrow, R.G.Gabriel y J.L.White. CLOS in Context-The Shape of the Design Space. En Object Oriented Programming – The CLOS Perspective. MIT Press, 1993

[BHD+98] Michael Bursell, Richard Hayton, Douglas Donaldson, Andrew Herbert. A Mobile Object Workbench. Second International Workshop on Mobile Agents 98 (MA'98). Stuttgart, Alemania, Septiembre de 1998

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, Larry Carter. “Distribution and Abstract Data Types in Emerald”. IEEE Transactions on Software Engineering, Vol. SE-13, nº 1, Enero 1987

[BLR96] M.N.Bouraqadi-Saâdani, T.Ledoux y F.Rivard. “Metaclass Composability”. En Proceedings of Composability Workshop. 10th European Conference on Object-Oriented Programming (ECOOP’96), Springer-Verlag, 1996.

[Boo94] Grady Booch. Object-Oriented Analysis and Design with Applications, 2nd edition. Benjamin Cummings. 1994. Versión en español: Análisis y diseño orientado a objetos con aplicaciones, 2ª edición. Addison-Wesley/Díaz de Santos. 1996.

[BRJ99] Grady Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide. Addison -Wesley. 1999.

[Caz98] W.Cazzola. “Evaluation of Object-Oriented Reflective Models”. En Object-Oriented Technology. ECOOP’98 Workshop Reader. S.Demeyer y J.Bosch, eds. Pág 386-387. Lecture Notes in Computer Science 1543. Springer-Verlag, 1998.

[CC91] R.S. Chin y S.T. Chanson. “Distributed Object-Based Programming Systems”. ACM Computing Surveys, V. 23, N.1. Marzo de 1991.

[CCI88] CCITT. Recommendation X.500: The Directory – Overview of Concepts, Models and Service. International Telecommunications Union, Place des Nations, 1211. Geneva, Switzerland. 1988.

[CDK94] George Coulouris, Jean Dollimore, Tim Kindberg. Distributed Systems. Concepts and Design (2ª edición). Addison-Wesley. 1994.

[Cha92] C.Chambers. The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. PhD Thesis. Stanford University. Marzo, 1992

[Che88] D. R. Cheriton. The V Distributed System. Communications of the ACM, Vol 31, nº 3, pp. 314-333. 1988.

[CIA+96] Juan Manuel Cueva Lovelle, Raúl Izquierdo Castanedo, Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, M. Angeles. Díaz Fondón, Fernando Álvarez García y Ana Belén Martínez Prieto. “Oviedo3: Acercando las tecnologías orientadas a objetos al hardware”. I Jornadas de trabajo en Ingeniería de Software. Sevilla. Noviembre de 1996.

[CIR+93] Roy H. Campbell, Nayeem Islam, David Raila y Peter W. Madany. Designing and Implementing Choices: an Object-Oriented System in C++”, Special Issue in Concurrent Object-Oriented Programming, Communications of the ACM (CACM), Vol. 36, nº 9, pp. 117-126, Septiembre de 1993

[CJM+91] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, Vincent F. Russo. Principles of Object-Oriented Operating System Design. Technical Report. University of Illinois at Urbana-Champaign. Julio de 1991.

[Coi87] P.Cointe. “MetaClasses are first class objects: the ObjVLisp model”. En Proceedings of OOPSLA’87, SIGPLAN Notices, Vol. 22, ACM, Orlando, Florida, Octubre 1987

[CS98] J.L.Contreras y J.-L.Sourrouille. “Adaptive Active Object”. En Object-Oriented Technology. ECOOP’98 Workshop Reader. S.Demeyer y J.Bosch, eds. Pág 369-371. Lecture Notes in Computer Science 1543. Springer-Verlag, 1998.

[CSI2000] Consejo Superior de Informática. Métrica Versión 3. En http://www.map.es/csi/

[DAM+90] P. Dasgupta, R. Chen, S. Menon, M. Person, R. Ananthanarayanan, U. Ramaschandran, M. Ahamad, R. LeBlanc, W. Applebe, J. Barnabeu-Auban, P. Hutto, M. Khalidi y C. Wilkenloh. “The Design and Implementation of the Clouds Distributed Operating System”. Computing Systems 3(1). 1990.

[Día2000] María Ángeles Díaz Fondón. Núcleo de seguridad para un sistema operativo orientado a objetos soportado por una máquina abstracta. Tesis Doctoral. Departamento de Informática, Universidad de Oviedo, Marzo de 2000

[DLA+91] P. Dasgupta, R.J. LeBlanc, M. Ahamad y U. Ramachandran. “The Clouds Distributed Operating System”. IEEE Computer, 24(11). 1991. Pág. 34-44.

[dRS84] J.des Rivieres y B.C.Smith. “The implementation of procedurally reflective languages”. En Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming. Pág. 331-347. Agosto 1984.

[Fer89] J.Ferber.”Computational Reflection in Class Based Object Oriented Languages”. En Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’1989). SIGPLAN Notices, 24:317-326, ACM 1989.

[Fer98] José Luis Fernández Díaz. Sistema de distribución de objetos en un entorno de máquinas abstractas orientadas a objetos. Proyecto Fin de Carrera 982027. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Junio de 1998

[FJ89] B.Foote y R.E.Johnson. “Reflective Facilities in Smalltalk-80”.En Proceedings of the Object-Oriented Programming, Languages, Systems and Applications (OOPSLA’89). ACM, 1989

[Foo90] B.Foote. “ Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?”. En ECOOP/OOPSLA’90 Workshop on Reflection and Metalevel Architectures.

[GC95] B.Gowing y V.Cahill. “Making Meta-Object Protocols Practical for Operating Systems”. En Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS’95), Lind, Suecia, 1995.

[GC96] B.Gowing y V.Cahill. “Meta-Object Protocols for C++: The Iguana Approach”. En Proceedings of the Reflection’96. Pp. 137-152. Abril 1996.

[Gie90] M.Gien. “Micro-Kernel Architecture Key to Modern Operating Systems Design”. UNIX REVIEW. 8(11). Noviembre, 1990.

[GK98a] M.Golm y J.Kleinöder. “MetaJava – A Platform for Adaptable Operating-System Mechanisms”. En 11th European Conference on Object-Oriented Programming (ECOOP’97). Workshop on Object-Orientation and Operating Systems. Junio 1997, Object-Oriented Technology. Lecture Notes in Computer Science 1357, pág. 507-514. I.Bosch, y S. Mitchell (Eds.) Springer-Verlag, 1998

[Gol97] M. Golm. “Design and Implementation of a Meta Architecture for Java”. Diplomarbeit im Fach Informatik. 1997

[Gos91] A. Goscinski. Distributed Operating Systems: The Logical Design. Addison-Wesley. 1991.

[Gra89] N.Graube. “MetaClass Compatibility”. En Proceedings of OOPSLA’89, SIGPLAN Notices, 24:305-316, ACM. New Orleans, Lousiana, Octubre 1989

[HBD+98] R. J. Hayton, M. H. Bursell, D. I. Donaldson, A. J. Herbert. Mobile Java Objects. Middleware 98. New York, EEUU. 1998.

[HBS96] Fritz Hohl, Joachim Baumann and Markus Straßer. *Beyond Java: Merging Corba-based Mobile Agents and WWW.* Joint W3C/OMG Workshop on Distributed Objects and Mobile Code, June 1996 Boston, Massachusetts.

[Hog94] Christine Hogan. The Tigger Cub Nucleus. Tesis Doctoral. Department of Computer Science, Trinity College, University of Dublin, Septiembre de 1994

[HR83] T. Härder y A. Reuter. Principles of Transaction-Oriented Database Recovery. Computing Surveys, Vol. 15, nº 4, 1983

[Hut87] Norman C. Hutchinson. Emerald: An Object-Based Language for Distributed Programming. Tesis Doctoral. Department of Computer Science, University of Washington, Seattle, Washington, Enero de 1987

[IBM96a] IBM Corporation, SOMobjects Developer´ Toolkit. Version 3.0. Programming Guide, Vol. 1 y 2. 1996

[IBM96b] IBM Corporation, SOMobjects Developer´ Toolkit. Version 3.0. Programming Reference, Vol. 1 a 4. 1996

[ISO92] International Standards Organization. Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use. ISO/IEC JTC1/SC212/WG17 CD 10746-1, International Standards Organization, 1992.

[Izq96] Raúl Izquierdo Castanedo. Máquina Abstracta Orientada a Objetos. Proyecto Fin de Carrera 9521I. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Septiembre de 1996.

[JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh. The Unified Software Development Process. Addison-Wesley. 1999.

[Jul88] Eric Jul. Object Mobility in a Distributed Object-Oriented System. Tesis Doctoral. Department of Computer Science, University of Washington, Seattle, Washington, Diciembre 1988

[KEL+62] T. Kilburn, D.B.G. Edwards, M.J. Lanigan y F.H. Summer. "One-level Storage System". *IRE Transactions on Electronic Computers*. Abril de 1962.

También en el capítulo 10 de D.P. Siewiorek, C.G. Bell y A. Newell. *Computer Structures: Principles and Examples*. McGraw-Hill. 1982.

[Kic92] G.Kiczales. "Towards a New Model of Abstraction in Software Engineering". En Proceedings of the International Workshop on New Models for Software Architecture'92; Reflection and Meta-Level Architectures. A. Yonezawa y B.C.Smith, Eds. Pág 1-11, Tokyo, Japón, 1992

[Kir97] Mary Kirtland. Object-Oriented Software Development Made Simple with COM+ Runtime Services. Microsoft Systems Journal, vol. 12, nº 11, Noviembre de 1997.

[KL93] G. Kiczales y J. Lamping. "Operating Systems: Why Object-Oriented?". En Proceedings of the Third International Workshop on Object-Orientation in Operating Systems. L.F. Cabrera y N. Hutchinson, eds. IEEE Computer Society Press, 1993. Pág. 25-30

[KR88] B. W. Kernighan, D. M. Ritchie. The C Programming Language. 2ª edición. Prentice Hall, Englewood Cliffs, 1988.

[Kra93] S. Krakowiak. "Issues in Object-Oriented Distributed Systems". International Conference in Decentralized and Distributed Systems, pp. 1-12, Palma de Mallorca, Septiembre de 1993.

[KRB91] G.Kickzales, J. desRivières y D.G.Bobrow. The art of MetaObject Protocol. MIT Press, Cambridge, Massachusetts, 1991

[KV92] J.L. Keedy y K. Vosseberg. "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System". En Proceedings of the *25th Hawaii International Conference on System Sciences*, Hawaii, EE.UU. 1992. Pág. 747-756.

[Led99] T.Ledoux. "OpenCorba: a Reflective Open Broker". En Proceedings of Reflection'99. LNCS, Springer-Verlag, 1999.

[LJP93] Rodger Lea, Chistian Jacquemot y Eric Pillevesse. COOL: System Support for Distributed Object-Oriented Programming. Technical Report CS/TR-93-68, Chorus systèmes. 1993.

[LMK+89] S. Leffler, M. McKusick, M. Karels y J. Quaterman. The design and implementation of the 4.3BSD Unix Operating System. Addison-Wesley, 1989

[LSAS+77] Barbara Liskov, Alan Snyder, Russell Atkinson y Craig Schaffert. Abstraction mechanisms in CLU. Communications of the ACM, 20(8):564-576, August 1977

[LYI95] R.Lea, Y.Yokote y J.-I.Itoh. "Adaptive operating system design using reflection". En Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V). Pp. 95-100. IEEE Computer Society Technical Committee on Operating Systems and Application Environments (TCOS). IEEE Computer Society Press, 1995

[Mae87] P.Maes. "Concepts and Experiments in Computational Reflection". En Proceedigns of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), SIGPLAN Notices, 22(12), 1987.

[Mae87] P.Maes. "Concepts and Experiments in Computational Reflection". En Proceedigns of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), SIGPLAN Notices, 22(12), 1987.

[Mae88] P.Maes. "Issues in Computational Reflection". Meta-Level Architectures and Reflection. Pp. 21-35. P.Maes y D.Nardi, Eds. Elsevier Science Publishers B.V. 1988

[McA93] J.McAffer. "The CodA MOP". En Proceedings of the OOPSLA'93 Reflection Workshop. 1993.

[McA95a] J.McAffer. "Meta-Level Programming with CodA". En Proceedings of the 9th Conference on Object-Oriented Programming, (ECOOP'95), LNCS 952, pág 190-214. Springer-Verlag, 1995.

[Mey88] Bertrand Meyer. Object-Oriented Software Construction. Prentice-Hall. 1988.

[Mey97] Bertrand Meyer. Object-oriented Software Construction (2ª edición). Prentice Hall. 1997.

[MGH+94] James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khaladi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, Sanjay R. Radia. An Overview of the Spring System. Proceedings of the CompCon Conference. San Francisco, California, 1994

[Mic95] Microsoft Corporation. The Component Object Model Specification. Microsoft, Octubre de 1995. Disponible en URL: http://www.microsoft.com/com.

[Mic98] Microsoft Corporation. Distributed Component Object Model Protocol. Microsoft, Enero de 1998. Disponible en URL: http://www.microsoft.com/com.

[MJD96] J.Malenfant, M.Jacques y F-N. Demers. "A Tutorial on Behavioral Reflection and its Implementation". En Proceedings of the Reflection'96 Conference, 1996.

[MMC95] P.Mulet, J.Malenfant y P.Cointe. "Towards a Methodology for Explicit Composition of MetaObjects". En Proceedings of OOPSLA'95. Pág 316-330. 1995

[Moc87] P. Mockapetris. Domain Names - Concepts and Facilities. Tech. Report RFC 1034, ftp://nic.ddn.mil/usr/pub/RFC. 1987.

[MPP86] B. P. Miller, D. L. Presotto, M. L. Powell. DEMOS/MP: The Development of a Distributed Operating Sytem. Computer Sciences Technical Report #650, Computer Sciences Department, University of Wisconsin-Madison, 1986.

[MR93] Patrick A. Muckelbauer, Vincent F. Russo. The Reinassance Distributed Object System. Technical Report TR.93-022, Department of Computer Science, Purdue University, 1993.

[MRT+90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren. Amoeba: A Distributed Operating System for the 1990's. IEEE Computer, vol. 23, pp 44-53, Mayo de 1990

[OMG99] Object Management Group. The Common Object Request Broker: Architecture and Specification, revision 2.3. Object Management Group. Junio de 1999. Disponible en URL: http://www.omg.org.

[Ort97] Francisco Ortín Soler. Diseño y Construcción del Sistema de Persistencia en Oviedo3. Proyecto Fin de Carrera 972001. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Septiembre de 1997.

[OSF92] Open Software Foundation. Introduction to OSF DCE. Open Software Foundation, Cambridge, USA, 1992

[PM83] M. L. Powell, B. P. Miller. Process Migration in DEMOS/MP. Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, pp. 110-119. 1983.

[PPD+90] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom. Plan 9 from Bell Labs. UKUUG Proceedings of the Summer 1990 Conf., London, England, 1990

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom. The Use of Name Spaces in Plan 9. Operating Systems Review, Vol. 27, nº 2, pp. 72-76, Abril de 1993

[PS75] D.L. Parnas and D.P. Siewiorek. Use of the concept of transparency in the

design of hierarchically structured systems. Communications of the ACM,

18(7):401-408, July 1975.

[RAA+92] M. Rozier, V. Abrossimov, F.Armand, I. Boule, M.Gien, M. Guillemont, F. Herman, C. Kaiser, S. Langlois, W. Neuhauser y P. Léonard. "Overview of the Chorus Distributed Operating System". En Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures. Francia. Abril de 1992. Pág. 39-69

[Riv88] J.des Rivières. "Control-Related Meta-Level Facilities in LISP". Meta-Level Architectures and Reflection. Pp. 201-109. P.Maes y D.Nardi, Eds. Elsevier Science Publishers B.V. 1988

[Rov86] Paul Rovner. Extending Modula-2 to build large integrated systems. IEEE

Software, 3(6):46-57, November 1986.

[RT74] D. M. Ritchie, K. Thompson. The UNIX Time-Sharing System. Communications of the ACM, vol. 17, nº. 7, pp. 365-375. 1974.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, Eric Jul. Emerald: A General-Purpose Programming Language. Software Practice and Experience, Vol. 21, nº 1, pp. 91-118, Enero de 1991

[SG98] A. Silberschatz, P.B. Galvin. Operating System Concepts (5ª edición). Addison-Wesley. 1998.

[SGH+89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, Celine Valot. SOS: An Object-Oriented Operating System - Assessment and Perspectives. Computing Systems, 2(4) 287-337, Otoño 1989

[Sha86] Marc Shapiro. "Structure and encapsulation in distributed systems: the proxy principle". 6th International Conference on Distributed Computer Systems, pp. 198-204, Cambridge, Mass. (USA), Mayo de 1986, IEEE.

[Sha90] Marc Shapiro. "Object-Support Operating Systems". Workshop on Operating Systems and Object Orientation at ECOOP-OOPSLA. Julio de 1990.

[SM93] Alexander B. Schill y Markus U. Mock. DC++: Distributed Object-Oriented System Support on top of OSF DCE. Distributed Systems Engineering, 1, pp. 112-125, 1993.

[SMF96] Alan C. Skousen, Donald S. Miller y Ronald G. Feigen. *The Sombrero Operating System: An Operating System for a Distributed Single Very Large Address Space – General Introduction*. Technical Report TR-96-005, Arizona State University, EE.UU.1996.

[Smi82] B.C.Smith. “Reflection and Semantics in a procedural language”. PhD Thesis. Technical Report 272 Massachusetts Institute of Technology. Laboratory for Computer Science. Cambridge, Massachusetts, 1982.

[Smi84] B.C.Smith. “Reflection and Semantics in Lisp”. En Proceedings of Principles of Programming Languages. 1984.

[Sta98] William Stallings. Operating Systems. Internals and Design Principles (3ª edición). Prentice Hall. 1998.

[Str92] B.Stroustrup. “Run Time Type Identification for C++”. En USENIX C++ Conference Proceedings, Portland, 1992.

[Sun97a] Sun Microsystems. Java Virtual Machine Specification. 1997.

[Sun97b] Sun Microsystems. Java Core Reflection, API and Specification. Febrero, 1997.

[Sun98] Sun Microsystems. Java Remote Method Invocation (RMI). En http://java.sun.com/products/jdk/rmi/index.html

[Taj2000] Lourdes Tajes Martínez. Modelo de computación concurrente para un sistema operativo orientado a objetos basado en una máquina abstracta. Tesis Doctoral. Departamento de Informática, Universidad de Oviedo, Marzo de 2000

[Taj97] L. Tajes Martínez. “Concurrency in object-oriented operating system”. 11th European Conference on Object-Oriented Programming (ECOOP’97) Jyväskylä (Finlandia) Junio, 1997

[TNO92] H.Tokuda, T.Nakajima y S.Oikawa. “Towards a New Operating System Architecture: Microkernel vs. Reflective Architecture”. En 10th Conference Proceedings, Japan Society for Software Science and Technology. 1992

[TRS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, Guido van Rossum. Experiences with the Amoeba Distributed Operating System. Communications of the ACM, vol. 33, nº 12, pp 46-63, Diciembre de 1990

[TV96] J. Tardo, L. Valente. *Mobile Agent Security and Telescrip*t. In IEEE CompCon, 1996.

[TW97] A.S. Tanenbaum, A.S. Woodhull Operating Systems: Design and Implementation (2ª Edición). Prentice Hall. 1997.

[TYT92] T. Tenma, Y. Yokote y M. Tokoro. “Implementing Persistent Objects in the Apertos Operating System”. En Proceedings of *the International Workshop on Object Orientation in Operating Systems (IWOOOS’92)*. Dourdan, Francia. Septiembre de 1992.

[Ven97] Bill Venners. *Solve real problems with aglets, a type of mobile agen*t. JavaWorld – Under the hood Magazine. Mayo 1997. Pp 2-4.

[VRH93] J. Vochteloo, S. Russell y G. Heiser. “Capability-Based Protection in the Mungi OS”. En Proceedings of the *IWOOOS'93*. Diciembre de 1993.

[Weg90] P. Wegner. “Concepts and Paradigms of Object-Oriented Programming”. OOPS Messenger. 1(1). Agosto 1990. Pág. 7-87

[WF88] M.Wand y D.P.Friedman. “The Mistery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower”. En Meta-Level Architectures and Reflection. P.Maes y D.Nardi, eds. Pag. 111-133. Elsevier Science Publishers. 1988

[WJ95] M. Wooldridge and N.R. Jennings, *Intelillent Agents: Theory and*

Practice. Enero 1995.

[WY88] T.Watanabe y A.Yonezawa. "Reflection in an Object-Oriented Concurrent Language". En Proceedigns of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88). SIGPLAN Notices, Vol 23, Pág 306-315. ACM Press, 1988.

[Yok92] Y.Yokote. "The Apertos Reflective Operating System: The Concept and Its Implementation". En Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92). Pp. 414-434. A.Paepcke, ed. ACM Special Interest Group on Programming Languages, ACM Press, October 1992. Also SIGPLAN Notices 27(10), October 1992

[YW89] A.Yonezawa y T.Watanabe. "An introduction to object-based reflective concurrent computation". En Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming. SIGPLAN Notices, 24:50-54. ACM Press, Abril, 1989.

[ZC96] C.Zimmermann y V.Cahill. "It's your choice – On the design and Implementation of a flexible Metalevel Archictecture". En Proceedings of the 3rd International Conference on Configurable Distributed Systems. IEEE Computer Society Press, Mayo 1996.