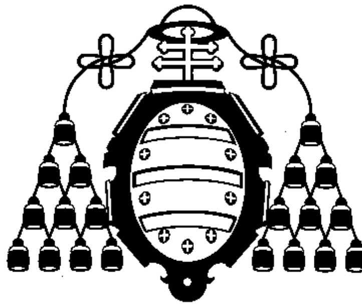
UNIVERSIDAD DE OVIEDO

Departamento de Informática



TESIS DOCTORAL

NÚCLEO DE SEGURIDAD PARA UN SISTEMA OPERATIVO ORIENTADO A OBJETOS SOPORTADO POR UNA MÁQUINA ABSTRACTA

Presentada por

María Ángeles Díaz Fondón

para la obtención del título de Doctor en Informática

Dirigida por el

Profesor Doctor D. Juan Manuel Cueva Lovelle

Oviedo, Enero de 2000

Resumen

Esta tesis describe un mecanismo de protección basado en capacidades diseñado para un Sistema Integral Orientado a Objetos (SIOO), que cumple los requisitos generales de diseño de un mecanismo de seguridad y los específicos para un SIOO. Además, aporta propiedades adicionales, como una protección automática de las capacidades y permisos de longitud variable. Todo ello configura un sistema que soluciona de manera más elegante y completa las necesidades de protección de un SIOO, con un rendimiento adecuado.

Los sistemas integrales orientados a objetos, basados únicamente en el paradigma de la orientación a objetos en todos sus componentes, y compuestos por una máquina abstracta reflectiva orientada a objeto extendida por un sistema operativo, solucionan el problema de la integración de las tecnologías orientadas a objetos sin los problemas de los sistemas actuales.

Estos sistemas se basan en la existencia de múltiples objetos distribuidos por el sistema que cooperan entre sí para llevar a cabo una tarea, posiblemente descargando bajo demanda código de sistemas remotos. En este contexto, es muy importante la existencia de un mecanismo de protección que posibilite la cooperación segura entre estas entidades.

Los sistemas existentes en estos momentos carecen de la uniformidad necesaria y no son válidos para Sistemas Integrales Orientados a Objetos (SIOO). Las soluciones aportadas (como las listas de control de acceso) no son lo suficientemente adecuadas como para resolver los problemas de seguridad que aparecen con el uso de la interoperabilidad de objetos distribuidos, puesto que incumplen numerosos requisitos generales de diseño de mecanismos de seguridad y específicos para SIOO.

La aproximación utilizada es el diseño de un nuevo modelo de protección: las capacidades orientadas a objetos, que consiste en esencia, en fundir las capacidades de manera uniforme en el modelo de objetos del sistema, lo que logra los requisitos de diseño impuestos, como el principio de mínimo privilegio, mediación total, uniformidad, homogeneidad, flexibilidad, etc. Se muestran ejemplos de sus ventajas, como la flexibilidad en la implementación de diferentes políticas de seguridad.

Para comprobar la factibilidad del modelo, se desarrolla una implementación del mismo partiendo de un prototipo de máquina abstracta del sistema integral. Sobre este prototipo se realizan análisis del rendimiento, que resulta adecuado.

Palabras clave

Protección, seguridad, mecanismo de protección, control de acceso, política de seguridad, capacidades, permisos, autoridad, autorización, sistemas integrales orientados a objetos, tecnología orientada a objetos, máquina abstracta, modelo de objetos, Oviedo3.

Abstract

This thesis describes a capability-based protection mechanism designed for an Integral Object-Oriented System (IOOS). It fulfills the general design requirements of a security mechanism, and IOOS specific requirements as well. Besides, additional properties are present, such as automatic protection of capabilities and variable-length permissions. This gives a system that solves protection needs of an IOOS in a more elegant and complete way, with adequate performance.

Integral object-oriented systems, based solely on the object-oriented paradigm for every component in the system, and structured with a reflective object-oriented abstract machine, extended by an operating system, solve the problem of the integration of object-oriented technologies without the problems associated with current systems.

These systems are based on the existence of many distributed objects, cooperating to achieve a common goal, possibly with on-demand downloading of remote code. In this context, having a protection mechanism that permits safe cooperation among these entities is crucial.

Current systems lack the necessary uniformity and are not valid for IOOS. Solutions such as access control lists are not adequate enough to solve the security problems that appear with distributed object interoperability, because many general and IOOS-specific requirements for a security mechanism are not fulfilled.

The approach taken is to design a new protection model: object-oriented capabilities. In essence, capabilities are merged with the object model of the system in a uniform way. This achieves the design requirements, such as the principle of least privilege, total mediation, uniformity, homogeneity, flexibility, etc. Examples of the advantages are show, such as a flexible implementation of different security policies.

To verify the viability of the model, an implementation of the model is carried upon a prototype of the IOOS abstract machine. A performance analysis is done on this implementation, resulting in an adequate performance.

Keywords

Protection, security, protection mechanism, access control, security policy, capabilities, permissions, authority, authorization, integral object-oriented systems, object-oriented technology, abstract machine, object model, Oviedo3.

Índice Resumido

1 OBJETIVOS Y ORGANIZACIÓN DE ESTA TESIS DOCTORAL 1

1.1 OBJETIVOS 1
1.2 EL PROYECTO DE INVESTIGACIÓN EN EL QUE ESTÁ ENMARCADA 2
1.3 FASES DE DESARROLLO 2
1.4 ORGANIZACIÓN DE LA MEMORIA....3

2 INTRODUCCIÓN....7

2.1 NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 7
2.2 SISTEMAS INTEGRALES ORIENTADOS A OBJETOS....9
2.3 SEGURIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS....11
2.4 ALGUNAS SOLUCIONES PLANTEADAS PARA EL CONTROL DE ACCESO....11

3 CONCEPTOS DE SEGURIDAD....15

3.1 INTRODUCCIÓN 15
3.2 ASPECTOS DE LA SEGURIDAD EN UN SISTEMA DE COMPUTACIÓN 16
3.3 ELEMENTOS DEL SISTEMA A LOS QUE AFECTA LA SEGURIDAD 18
3.4 EL PROBLEMA DEL CONTROL DE ACCESO (PROTECCIÓN)....18
3.5 CONCEPTOS DE SEGURIDAD Y PROTECCIÓN....20
3.6 TAXONOMÍAS DE SEGURIDAD....21
3.7 EVOLUCIÓN DEL CONTROL DE ACCESO A RECURSOS Y SERVICIOS 23
3.8 DOMINIOS DE PROTECCIÓN 27
3.9 PRINCIPIOS DE DISEÑO DE MECANISMOS DE SEGURIDAD 29
3.10 PROBLEMAS QUE DEBE RESOLVER UN SISTEMA DE SEGURIDAD....31

4 MODELOS DE SEGURIDAD 41

4.1 MODELO DE LA MATRIZ DE ACCESO....41
4.2 MODELO DE SEGURIDAD TAKE-GRANT....42
4.3 MODELOS DE SEGURIDAD DE FLUJO DE INFORMACIÓN....42
4.4 NÚCLEO DE SEGURIDAD....46
4.5 IMPLEMENTACIONES Y VARIANTES DE LA MATRIZ DE CONTROL DE ACCESO 46

5 RESUMEN DE LA ARQUITECTURA DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 53

5.1 REQUISITOS DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS....53
5.2 MAQUINA ABSTRACTA OO + SISTEMA OPERATIVO OO = SISTEMA INTEGRAL OO....55
5.3 MODELO ÚNICO DE OBJETOS DEL SISTEMA INTEGRAL 58
5.4 ARQUITECTURA DE REFERENCIA DE LA MÁQUINA ABSTRACTA 59
5.5 EXTENSIÓN DE LA MÁQUINA ABSTRACTA PARA PROPORCIONAR LA FUNCIONALIDAD DEL SISTEMA OPERATIVO. REFLECTIVIDAD 62

6 REQUISITOS DE UN MECANISMO BÁSICO DE PROTECCIÓN PARA UN SISTEMA INTEGRAL ORIENTADO A OBJETOS....65

6.1 MECANISMO DE PROTECCIÓN PARA UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 65
6.2 REQUISITOS DE PROTECCIÓN....66
6.3 CUMPLIMIENTO DE LOS PRINCIPIOS BÁSICOS DE DISEÑO DE MECANISMOS 67

7 PANORAMA DE SEGURIDAD EN ALGUNOS SISTEMAS OPERATIVOS 69

7.1 SISTEMA OPERATIVO AMADEUS 69
7.2 SISTEMA OPERATIVO AMOEBA 72
7.3 SISTEMA OPERATIVO CHOICES 76
7.4 SISTEMA OPERATIVO GUIDE....79

7.5 Sistema Operativo Grasshopper ... 82
7.6 Sistema Operativo Hydra ... 84
7.7 Sistema Operativo KeyKOS ... 87
7.8 Sistema Operativo Mungi ... 89
7.9 Sistema Operativo Opal ... 92
7.10 Sistema Operativo Spin ... 94
7.11 Tabla resumen de las principales características de los sistemas operativos analizados ... 96
7.12 Conclusiones sobre los sistemas estudiados ... 97

8 SEGURIDAD EN LA PLATAFORMA JAVA ... 99

8.1 Evolución de Internet ... 99
8.2 Plataforma Java ... 99
8.3 El problema de la protección ... 99
8.4 Tratamiento de la seguridad en Java ... 100
8.5 Modelo inicial de seguridad en Java ... 101
8.6 Crítica a la seguridad en Java ... 109

9 ELECCIÓN DE CAPACIDADES COMO MECANISMO BASE DE PROTECCIÓN ... 113

9.1 Análisis del uso de listas de control de acceso ... 113
9.2 Análisis del uso de Capacidades ... 119
9.3 Análisis del modelo mixto ... 122
9.4 Análisis del modelo de retículo de seguridad ... 122
9.5 Introspección en pila ... 123
9.6 Comportamiento de los modelos respecto a los problemas de seguridad más comunes de un sistema ... 123
9.7 Elección de Capacidades como mecanismo de protección ... 130

10 ESTUDIO DE CAPACIDADES ... 133

10.1 Conceptos generales sobre capacidades ... 133
10.2 Implementación de Capacidades ... 135
10.3 Permisos sobre los métodos que portan las capacidades ... 137
10.4 Permisos relativos a la propia capacidad ... 138
10.5 Operaciones intrínsecas a las capacidades ... 139
10.6 La tecnología de orientación a objetos y las capacidades ... 140
10.7 Implementación del mecanismo de comprobación de permisos ... 140
10.8 Ventajas de las capacidades en general ... 143
10.9 Inconvenientes de las capacidades en general ... 144

11 DISEÑO DEL MODELO DE PROTECCIÓN ... 147

11.1 Diseño de las Capacidades: Capacidades Orientadas a Objetos ... 147
11.2 Diseño del mecanismo de comprobación de permisos: implementación en la máquina abstracta ... 156
11.3 Resumen de las características del modelo de protección ... 158
11.4 Resumen del modo de operación del mecanismo ... 158

12 VENTAJAS DEL MODELO DE PROTECCIÓN DISEÑADO ... 161

12.1 Cumplimiento de los principios básicos de diseño de mecanismos de seguridad ... 161
12.2 Cumplimiento de los requisitos de protección para un sistema integral orientado a objetos ... 162
12.3 Resolución de problemas de un sistema de seguridad ... 164
12.4 Ventajas adicionales ... 165

13 POLÍTICAS DE SEGURIDAD SOBRE EL MECANISMO BÁSICO ... 167

13.1 Estructura general del sistema. ... 168
13.2 Políticas de Seguridad ... 169

14 ESPECIFICACIÓN DE LA MÁQUINA ABSTRACTA SOBRE LA QUE SE IMPLANTA EL MECANISMO BÁSICO DE PROTECCIÓN ... 177

14.1 El sistema Integral Orientado a Objetos Oviedo3 ... 177
14.2 La máquina abstracta Carbayonia ... 178

14.3 El lenguaje Carbayón 180

15 DISEÑO E IMPLEMENTACIÓN DEL PROTOTIPO INICIAL DE LA MÁQUINA ABSTRACTA.. 189

15.1 Introducción 189
15.2 Diseño orientado a objetos 191
15.3 Implementación primitiva de elementos básicos 192
15.4 Representación de los elementos de la máquina 193
15.5 Funcionamiento general de la máquina 196
15.6 Creación de elementos 197
15.7 Ejecución de instrucciones 199

16 IMPLANTACIÓN DEL MECANISMO DE PROTECCIÓN EN LA MÁQUINA ABSTRACTA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS..205

16.1 Integración de Capacidades con referencias 205
16.2 Implementación de la instrucción de restricción de permisos: ForbidExec..207
16.3 Comprobación de permisos: modificación de la invocación de métodos (Instrucción Call) .210
16.4 Modificación de la instrucción Assign 212
16.5 Modificación de la instrucción New..213
16.6 Modificación de la instrucción Delete 214

17 ANÁLISIS DEL RENDIMIENTO DEL SISTEMA CON LA INCORPORACIÓN DEL MECANISMO DE PROTECCIÓN 215

17.1 Creación de una Clase primitiva Reloj para evaluar el rendimiento 216
17.2 Pruebas de rendimiento 216
17.3 Consideraciones sobre el rendimiento 220

18 ELABORACIÓN DE UN ENTORNO OPERATIVO SOBRE LA MÁQUINA ABSTRACTA 225

18.1 Introducción 225
18.2 Servicio de seguridad 225
18.3 Entorno Operativo..226
18.4 Requisitos de diseño del Entorno Operativo 226
18.5 Definición de un enlace del usuario con la máquina abstracta. Clase System......................227
18.6 Otras modificaciones..228
18.7 Características generales del Entorno Operativo 229
18.8 Órdenes del Shell 230
18.9 Diseño e Implementación del Entorno Operativo 234
18.10 Diseño detallado 235
18.11 Documentación del Entorno..237

19 APLICACIÓN A OTROS SISTEMAS Y TRABAJO RELACIONADO 241

19.1 Aplicación a la plataforma Java 241
19.2 Trabajo relacionado 242

20 CONCLUSIONES 245

20.1 Capacidades Orientadas a Objetos 245
20.2 Algunos resultados destacables 247
20.3 Trabajo y líneas de investigación futuras 250

BIBLIOGRAFÍA..253

Índice General

1 OBJETIVOS Y ORGANIZACIÓN DE ESTA TESIS DOCTORAL 1

1.1 OBJETIVOS 1
1.2 EL PROYECTO DE INVESTIGACIÓN EN EL QUE ESTÁ ENMARCADA 2
1.3 FASES DE DESARROLLO 2
1.4 ORGANIZACIÓN DE LA MEMORIA......................................3

2 INTRODUCCIÓN......................................7

2.1 NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 7
2.1.1 El problema de la desadaptación de impedancias 7
2.1.1.1 Comunicación de alto nivel entre objetos situados en espacios de direcciones diferentes 8
2.1.1.2 Desadaptación de interfaces......................................8
2.1.2 El problema de la interoperabilidad entre modelos de objetos......................................8
2.1.3 Problemas de las capas de adaptación sobre sistemas tradicionales 9
2.2 SISTEMAS INTEGRALES ORIENTADOS A OBJETOS......................................9
2.2.1 Requisitos de un sistema integral orientado a objetos 10
2.2.1.1 Uniformidad conceptual en torno a la orientación a objetos 10
2.2.1.2 Transparencia de distribución y persistencia......................................10
2.2.1.3 Heterogeneidad y portabilidad 10
2.2.1.4 Seguridad 10
2.2.1.5 Concurrencia......................................10
2.2.1.6 Multilenguaje/Interoperabilidad......................................10
2.2.1.7 Flexibilidad......................................11
2.2.2 Arquitectura de un Sistema Integral Orientado a Objetos 11
2.3 SEGURIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS......................................11
2.4 ALGUNAS SOLUCIONES PLANTEADAS PARA EL CONTROL DE ACCESO 11
2.4.1 Inconvenientes de los planteamientos adoptados......................................12
2.4.2 Núcleo de seguridad para un sistema integral Orientado a Objetos 13

3 CONCEPTOS DE SEGURIDAD......................................15

3.1 INTRODUCCIÓN 15
3.2 ASPECTOS DE LA SEGURIDAD EN UN SISTEMA DE COMPUTACIÓN 16
3.2.1 Seguridad en el acceso al sistema: Autenticación de usuarios 16
3.2.2 Seguridad en el uso de recursos y servicios 16
3.2.3 Seguridad en el uso de redes......................................17
3.3 ELEMENTOS DEL SISTEMA A LOS QUE AFECTA LA SEGURIDAD 18
3.4 EL PROBLEMA DEL CONTROL DE ACCESO (PROTECCIÓN)......................................18
3.4.1 Objetivo de esta tesis doctoral......................................18
3.4.2 Aspectos que abarca el control de acceso......................................19
3.4.2.1 Prevención del acceso......................................19
3.4.2.2 Limitación del acceso 19
3.4.2.3 Concesión de acceso 19
3.4.2.4 Revocación de acceso 19
3.5 CONCEPTOS DE SEGURIDAD Y PROTECCIÓN......................................20
3.6 TAXONOMÍAS DE SEGURIDAD......................................21
3.6.1 Clasificación en niveles de seguridad del Departamento de Defensa de EE.UU 21
3.6.2 Tipos de control de acceso 22
3.7 EVOLUCIÓN DEL CONTROL DE ACCESO A RECURSOS Y SERVICIOS 23
3.7.1 Ninguna protección 23
3.7.2 Protección de los recursos básicos del sistema......................................23
3.7.2.1 Protección de Memoria......................................23
3.7.2.2 Protección de Ficheros......................................25
3.7.3 Protección de servicios generales del sistema y de los propios usuarios......................................25
3.7.4 Situación actual: control heterogéneo 26

3.7.4.1 Problemas de la situación actual 26
3.8 Dominios de protección 27
3.8.1 Contextos de desarrollo de un dominio (cambios de dominio) *28*
3.8.2 Dominios en el sistema operativo Unix *28*
3.9 Principios de diseño de mecanismos de seguridad 29
3.9.1 Mínimo privilegio *29*
3.9.1.1 Control de "Caballos de Troya" 29
3.9.2 Ahorro de mecanismos *30*
3.9.2.1 Diseño del mecanismo de seguridad como parte integral del sistema 30
3.9.3 Aceptación *30*
3.9.3.1 Mecanismo de seguridad conceptualmente sencillo 30
3.9.3.2 Mecanismo de seguridad fácil de usar 30
3.9.4 Mediación total *31*
3.9.4.1 Imposibilidad de esquivar el mecanismo de seguridad 31
3.9.5 Diseño abierto *31*
3.9.5.1 Peligro del probable descubrimiento de un diseño secreto 31
3.10 Problemas que debe resolver un sistema de seguridad 31
3.10.1 El problema de la fuga de información *32*
3.10.2 Defensa perimetral *34*
3.10.3 El representante confuso *35*
3.10.4 Confinamiento *36*
3.10.5 Conspiradores en comunicación *37*
3.10.6 Otros puntos de vista de la fuga de información y de los problemas que debe resolver un sistema de seguridad *37*
3.10.6.1 Propagación 37
3.10.6.2 Sospecha mutua 38
3.10.6.3 Delegación 38
3.10.6.4 Revocación 38
3.10.6.5 Confinamiento 39
3.10.6.6 Caballos de Troya 39

4 MODELOS DE SEGURIDAD 41

4.1 Modelo de la matriz de acceso 41
4.2 Modelo de seguridad Take-Grant 42
4.3 Modelos de seguridad de flujo de información 42
4.4 Núcleo de seguridad 46
4.5 Implementaciones y variantes de la matriz de control de acceso 46
4.5.1 Listas de control de acceso *47*
4.5.1.1 Inconvenientes 47
4.5.1.2 Ventajas 48
4.5.1.3 Grupos de protección 48
4.5.1.4 Propagación de los permisos de acceso 48
4.5.2 Capacidades *49*
4.5.2.1 Falsificación de capacidades 50
4.5.2.2 Ventajas 50
4.5.2.3 Inconvenientes 50
4.5.3 Mixta (Clave-permisos) *51*

5 RESUMEN DE LA ARQUITECTURA DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 53

5.1 Requisitos de un Sistema Integral Orientado a Objetos 53
5.1.1 Uniformidad conceptual en torno a la orientación a objetos *54*
5.1.2 Modo de trabajo exclusivamente orientado a objetos *54*
5.1.3 Homogeneidad de objetos *54*
5.1.4 Transparencia *54*
5.1.5 Heterogeneidad y portabilidad *55*
5.1.6 Seguridad *55*
5.1.7 Concurrencia *55*
5.1.8 Multilenguaje / Interoperabilidad *55*
5.1.9 Flexibilidad *55*
5.2 Maquina abstracta OO + Sistema Operativo OO = Sistema Integral OO 55
5.2.1 Máquina Abstracta Orientada a Objetos *56*

5.2.1.1 Modelo único de objetos 56
5.2.1.2 Reflectividad 56
5.2.2 Sistema Operativo Orientado a Objetos 56
5.2.2.1 Transparencia: persistencia, distribución, concurrencia y protección 56
5.2.3 Orientación a objetos 56
5.2.4 Espacio único de objetos sin separación usuario/sistema 56
5.2.5 Identificador de objetos único, global e independiente 57
5.3 MODELO ÚNICO DE OBJETOS DEL SISTEMA INTEGRAL 58
5.3.1 Parte del modelo en la máquina abstracta 58
5.3.2 Parte del modelo en el sistema operativo 58
5.4 ARQUITECTURA DE REFERENCIA DE LA MÁQUINA ABSTRACTA 59
5.4.1 Propiedades fundamentales de una máquina abstracta para un SIOO 59
5.4.2 Principios de diseño de la máquina 59
5.4.3 Estructura de referencia 59
5.4.4 Juego de instrucciones 61
5.4.4.1 Instrucciones declarativas 61
5.4.4.2 Instrucciones de comportamiento 61
5.4.5 Ventajas del uso de una máquina abstracta 61
5.5 EXTENSIÓN DE LA MÁQUINA ABSTRACTA PARA PROPORCIONAR LA FUNCIONALIDAD DEL SISTEMA OPERATIVO. REFLECTIVIDAD 62
5.5.1 Modificación interna de la máquina. 62
5.5.2 Extensión de la funcionalidad de las clases básicas 62
5.5.3 Colaboración con el funcionamiento de la máquina. Reflectividad. 62

6 REQUISITOS DE UN MECANISMO BÁSICO DE PROTECCIÓN PARA UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 65

6.1 MECANISMO DE PROTECCIÓN PARA UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 65
6.1.1 Plataforma de desarrollo del mecanismo de protección 65
6.1.2 Seguridad en el uso de recursos y servicios 65
6.1.3 Mecanismo de protección y política de seguridad 65
6.2 REQUISITOS DE PROTECCIÓN 66
6.2.1 Flexibilidad 66
6.2.2 Movilidad 66
6.2.3 Uniformidad en la Orientación a Objetos 67
6.2.4 Homogeneidad 67
6.2.5 Protección de granularidad fina 67
6.3 CUMPLIMIENTO DE LOS PRINCIPIOS BÁSICOS DE DISEÑO DE MECANISMOS 67

7 PANORAMA DE SEGURIDAD EN ALGUNOS SISTEMAS OPERATIVOS 69

7.1 SISTEMA OPERATIVO AMADEUS 69
7.1.1 Características generales de Amadeus 69
7.1.2 Mecanismo de protección 69
7.1.3 Crítica del mecanismo de protección 70
7.2 SISTEMA OPERATIVO AMOEBA 72
7.2.1 Características generales del sistema 72
7.2.2 Mecanismo de protección 72
7.2.2.1 Capacidades y puertos 72
7.2.2.2 Estructura de las capacidades 73
7.2.2.3 Modo de trabajo: capacidades de usuario encriptadas 73
7.2.2.4 Política de seguridad: servicio de denominación y de directorios 74
7.2.3 Crítica del mecanismo de protección 74
7.3 SISTEMA OPERATIVO CHOICES 76
7.3.1 Características generales del sistema 76
7.3.2 Mecanismo de protección 76
7.3.3 Crítica del mecanismo de protección 77
7.4 SISTEMA OPERATIVO GUIDE 79
7.4.1 Características generales del sistema Guide 79
7.4.2 Mecanismo de protección 79
7.4.2.1 Vistas 79
7.4.2.2 Protección en el enlace dinámico 80
7.4.2.3 Resolución del problema de la delegación 80

7.4.3 Crítica del mecanismo de protección ... *80*
7.5 SISTEMA OPERATIVO GRASSHOPPER ... 82
7.5.1 Características generales del sistema ... *82*
7.5.2 Mecanismo de protección ... *82*
7.5.2.1 Capacidades segregadas para proteger contenedores y *loci* ... 82
7.5.2.2 Información que poseen las entidades a proteger ... 82
7.5.2.3 Modo de funcionamiento: mixto capacidades / listas de control de acceso ... 82
7.5.3 Crítica del mecanismo de protección ... *83*
7.6 SISTEMA OPERATIVO HYDRA ... 84
7.6.1 Características Generales ... *84*
7.6.2 Mecanismo de protección ... *84*
7.6.2.1 Parámetros plantilla de los procedimientos ... 85
7.6.3 Crítica del mecanismo de protección ... *85*
7.7 SISTEMA OPERATIVO KEYKOS ... 87
7.7.1 Características generales del sistema ... *87*
7.7.2 Mecanismo de protección ... *87*
7.7.2.1 Estructura de las capacidades ... 87
7.7.2.2 Monitor de referencia ... 87
7.7.3 Crítica del mecanismo de protección ... *88*
7.8 SISTEMA OPERATIVO MUNGI ... 89
7.8.1 Características generales del sistema ... *89*
7.8.2 Mecanismo de protección ... *89*
7.8.2.1 Capacidades *password* ... 89
7.8.2.2 Sistema jerárquico de seguridad ... 90
7.8.3 Crítica del mecanismo de protección ... *90*
7.9 SISTEMA OPERATIVO OPAL ... 92
7.9.1 Características generales del sistema ... *92*
7.9.2 Mecanismo de protección ... *92*
7.9.2.1 Dominios de protección ... 92
7.9.2.2 Portales ... 92
7.9.3 Crítica del mecanismo de protección ... *93*
7.10 SISTEMA OPERATIVO SPIN ... 94
7.10.1 Características Generales ... *94*
7.10.2 Crítica del mecanismo de protección ... *94*
7.11 TABLA RESUMEN DE LAS PRINCIPALES CARACTERÍSTICAS DE LOS SISTEMAS OPERATIVOS ANALIZADOS ... 96
7.12 CONCLUSIONES SOBRE LOS SISTEMAS ESTUDIADOS ... 97

8 SEGURIDAD EN LA PLATAFORMA JAVA ... 99

8.1 EVOLUCIÓN DE INTERNET ... 99
8.2 PLATAFORMA JAVA ... 99
8.3 EL PROBLEMA DE LA PROTECCIÓN ... 99
8.4 TRATAMIENTO DE LA SEGURIDAD EN JAVA ... 100
8.4.1 Seguridad en el diseño del lenguaje ... *100*
8.4.2 Verificador de bytecodes ... *100*
8.4.3 Cargador de clases ... *100*
8.4.4 Protección del sistema de ficheros y del acceso a la red ... *100*
8.5 MODELO INICIAL DE SEGURIDAD EN JAVA ... 101
8.5.1 Política de seguridad: Caja de arena ... *101*
8.5.1.1 Mecanismo de protección: Clase SecurityManager ... 102
8.5.2 Ampliaciones al modelo original: Firmas digitales (JDK 1.1) ... *102*
8.5.2.1 Firma digital ... 102
8.5.2.2 Política de seguridad de la caja de arena ampliada con firmas digitales ... 103
8.5.2.3 Mecanismo de protección ... 103
8.5.3 Nueva arquitectura de seguridad en Java2 (JDK 1.2) ... *104*
8.5.3.1 Política de seguridad ... 104
8.5.3.2 Dominios de protección ... 104
8.5.3.3 Permisos de acceso ... 106
8.5.3.4 Control de acceso a recursos ... 106
8.5.3.5 Resumen de la integración de seguridad en el JDK1.2 ... 108
8.6 CRÍTICA A LA SEGURIDAD EN JAVA ... 109
8.6.1 Inconvenientes de la caja de arena ... *109*
8.6.1.1 Inflexibilidad por la granularidad gruesa y falta del principio de mínimo privilegio ... 109

8.6.2 Inconvenientes de la protección basada en firmas digitales 109
8.6.2.1 Suposición de buena voluntad de los proveedores 109
8.6.2.2 No es un mecanismo de protección 110
8.6.2.3 Limitación de la expansión del mercado de software 110
8.6.3 Inconvenientes de la plataforma Java con la nueva arquitectura 110
8.6.3.1 Base de computación fiable muy grande y compleja 110
8.6.3.2 Granularidad no al nivel de objetos individuales 110
8.6.3.3 Falta de uniformidad y generalidad del mecanismo: responsabilidad del usuario en la implementación de la protección 111
8.6.3.4 Falta de flexibilidad para introducir protección a posteriori 111
8.6.3.5 Falta de adaptabilidad. Mecanismo "pesado" 111
8.6.3.6 Sin mediación total 111
8.6.3.7 Aceptación difícil por el usuario 111

9 ELECCIÓN DE CAPACIDADES COMO MECANISMO BASE DE PROTECCIÓN 113

9.1 ANÁLISIS DEL USO DE LISTAS DE CONTROL DE ACCESO 113
9.1.1 Falta de escalabilidad 114
9.1.1.1 Sistemas con gran número de objetos: problemas de escalabilidad en el espacio y en el tiempo 114
9.1.1.2 Problemas de escalabilidad en la movilidad de objetos 115
9.1.2 Intento de reducción de los problemas de escalabilidad 115
9.1.2.1 Reducción del número de LCA: agrupación en ámbitos 115
9.1.2.2 Reducción del número de entradas de cada LCA: usuarios y grupos 115
9.1.3 Problemas adicionales debidos a la falta de escalabilidad 116
9.1.3.1 Falta de uniformidad 116
9.1.3.2 Falta de granularidad fina: mediación parcial sin mínimo privilegio 117
9.1.3.3 Poca flexibilidad 117
9.1.4 Otros problemas de las listas de control de acceso 118
9.1.4.1 Sobrecarga del mecanismo de protección 118
9.1.4.2 Dependencia entre objetos 119
9.1.4.3 Complejidad 119
9.2 ANÁLISIS DEL USO DE CAPACIDADES 119
9.2.1 Definición de capacidades 119
9.2.2 Fundamento de las capacidades 120
9.2.3 Control de granularidad fina 120
9.2.3.1 Mejora de la escalabilidad en el espacio y en el tiempo 121
9.2.4 Independencia entre objetos respecto al mecanismo de protección 121
9.2.5 Integración en el modelo de objetos 121
9.2.6 Flexibilidad 122
9.3 ANÁLISIS DEL MODELO MIXTO 122
9.4 ANÁLISIS DEL MODELO DE RETÍCULO DE SEGURIDAD 122
9.5 INTROSPECCIÓN EN PILA 123
9.6 COMPORTAMIENTO DE LOS MODELOS RESPECTO A LOS PROBLEMAS DE SEGURIDAD MÁS COMUNES DE UN SISTEMA 123
9.6.1 El problema de la propagación (conspiradores en comunicación) 123
9.6.1.1 Propagación indirecta: la ilusión de la solución del problema de la propagación con listas de control de acceso 124
9.6.1.2 Expresión de la prohibición de propagación 125
9.6.1.3 Irresolubilidad del problema de la propagación 125
9.6.2 El problema de la revocación 125
9.6.2.1 La ilusión de la revocación selectiva con listas de control de acceso 125
9.6.2.2 Revocación no selectiva de capacidades 126
9.6.2.3 Revocación selectiva con capacidades: fachadas 126
9.6.2.4 Extensión del monitor de referencia para la revocación selectiva 127
9.6.3 El problema de la delegación (representante confuso) 128
9.6.4 El problema del confinamiento 129
9.6.4.1 Confinamiento con capacidades: monitor de referencia 129
9.7 ELECCIÓN DE CAPACIDADES COMO MECANISMO DE PROTECCIÓN 130
9.7.1 Integración en el modelo de objetos como característica destacable 130

10 ESTUDIO DE CAPACIDADES 133

10.1 CONCEPTOS GENERALES SOBRE CAPACIDADES 133
10.1.1 Qué es una capacidad 133
10.1.2 Modo de operación 134

10.1.2.1 Expresividad 134
10.1.3 Obtención de capacidades *134*
10.1.4 Protección de las capacidades *135*
10.1.4.1 Falsificación de una capacidad 135
10.2 IMPLEMENTACIÓN DE CAPACIDADES 135
10.2.1 Etiquetado hardware *135*
10.2.2 Segregación *136*
10.2.3 Capacidades dispersas *136*
10.3 PERMISOS SOBRE LOS MÉTODOS QUE PORTAN LAS CAPACIDADES 137
10.3.1 Todo/Nada *137*
10.3.2 Número fijo de permisos *137*
10.4 PERMISOS RELATIVOS A LA PROPIA CAPACIDAD 138
10.4.1.1 Permiso de copia de la capacidad 138
10.4.1.2 Permiso de eliminación de instancias 138
10.4.1.3 Permiso de eliminación de la propia capacidad 138
10.5 OPERACIONES INTRÍNSECAS A LAS CAPACIDADES 139
10.5.1.1 Restricción de permisos 139
10.5.1.2 Duplicación (copia) de capacidades 139
10.5.1.3 Creación de capacidades 139
10.5.1.4 Inicialización de capacidades 139
10.5.1.5 Revocación de capacidades 140
10.6 LA TECNOLOGÍA DE ORIENTACIÓN A OBJETOS Y LAS CAPACIDADES 140
10.7 IMPLEMENTACIÓN DEL MECANISMO DE COMPROBACIÓN DE PERMISOS 140
10.7.1 Integración el núcleo del sistema operativo *140*
10.7.2 Integración en servidores del usuario *141*
10.7.2.1 Inconvenientes la integración en servidores de usuario 141
10.7.3 Integración al nivel de compilador *142*
10.7.4 Integración al nivel del soporte en tiempo de ejecución del lenguaje *142*
10.8 VENTAJAS DE LAS CAPACIDADES EN GENERAL 143
10.8.1 Doble función de las capacidades: Denominación y protección *143*
10.8.2 Independencia cliente/servidor *143*
10.8.2.1 Eficiencia 143
10.8.2.2 Simplicidad 143
10.8.2.3 Flexibilidad 143
10.8.2.4 Flexibilidad en el número de paradigmas diferentes que pueden ser implementados 143
10.8.2.5 Escalabilidad 143
10.9 INCONVENIENTES DE LAS CAPACIDADES EN GENERAL 144
10.9.1.1 Control de propagación 144
10.9.1.2 Revisión (trazabilidad) 144
10.9.1.3 Revocación de permisos de acceso 145
10.9.1.4 Recolección de basura 145

11 DISEÑO DEL MODELO DE PROTECCIÓN 147

11.1 DISEÑO DE LAS CAPACIDADES: CAPACIDADES ORIENTADAS A OBJETOS 147
11.1.1 Protección de las capacidades: Nuevo tipo de capacidades: Orientadas a Objetos *147*
11.1.1.1 Capacidades Orientadas a Objetos 148
11.1.1.2 Referencias como capacidades básicas 149
11.1.1.3 Ampliación de la información de las referencias 149
11.1.2 Permisos sobre los métodos que portan las capacidades: Número de permisos variable *150*
11.1.2.1 Permisos sobre todos los métodos de un objeto: poco flexible 150
11.1.2.2 Número fijo de permisos: demasiado restrictivo 150
11.1.2.3 Capacidades Orientadas a Objetos: Número de permisos variable 151
11.1.3 Permisos relativos a la propia capacidad: Salto de protección *151*
11.1.3.1 Permiso de copia: no necesario 151
11.1.3.2 Permiso de eliminación de instancias: protección de un método normal de todos los objetos 152
11.1.3.3 Permiso de eliminación de la propia capacidad: no necesario 152
11.1.3.4 Capacidades Orientadas a Objetos: Salto de la protección 153
11.1.4 Operaciones intrínsecas a las capacidades *154*
11.1.4.1 Copia de capacidades: mecanismo ya existente 154
11.1.4.2 Creación de las capacidades: mecanismo ya existente 154
11.1.4.3 Inicialización de capacidades: mecanismo ya existente 155
11.1.4.4 Restricción de capacidades: operación única “Restringir <capacidad>, <método>” 155
11.1.4.5 Revocación selectiva: no es necesario incluirla en el mecanismo básico 156

11.2 DISEÑO DEL MECANISMO DE COMPROBACIÓN DE PERMISOS: IMPLEMENTACIÓN EN LA MÁQUINA ABSTRACTA 156
11.2.1 Capacidades Orientadas a Objetos: Implementación en la máquina abstracta 157
11.3 RESUMEN DE LAS CARACTERÍSTICAS DEL MODELO DE PROTECCIÓN 158
11.4 RESUMEN DEL MODO DE OPERACIÓN DEL MECANISMO 158

12 VENTAJAS DEL MODELO DE PROTECCIÓN DISEÑADO 161

12.1 CUMPLIMIENTO DE LOS PRINCIPIOS BÁSICOS DE DISEÑO DE MECANISMOS DE SEGURIDAD 161
12.1.1 Mínimo privilegio 161
12.1.2 Ahorro de mecanismos: robustez 161
12.1.3 Aceptación 162
12.1.4 Mediación total 162
12.1.5 Diseño abierto 162
12.2 CUMPLIMIENTO DE LOS REQUISITOS DE PROTECCIÓN PARA UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 162
12.2.1 Flexibilidad 162
12.2.2 Movilidad 163
12.2.3 Uniformidad en la Orientación a objetos 163
12.2.4 Homogeneidad 163
12.2.5 Protección de granularidad fina 163
12.3 RESOLUCIÓN DE PROBLEMAS DE UN SISTEMA DE SEGURIDAD 164
12.3.1 Defensa perimetral 164
12.3.2 Delegación (representante confuso) 164
12.3.3 Confinamiento 164
12.4 VENTAJAS ADICIONALES 165
12.4.1 Protección automática de capacidades 165
12.4.2 Permisos de longitud arbitraria 165
12.4.3 Extensibilidad del sistema con seguridad y flexibilidad 166
12.4.4 Abstracciones adicionales para la protección innecesarias 166
12.4.4.1 Concepto de superusuario innecesario: sistemas más seguros 166

13 POLÍTICAS DE SEGURIDAD SOBRE EL MECANISMO BÁSICO 167

13.1 ESTRUCTURA GENERAL DEL SISTEMA 168
13.1.1 Objeto servidor de nombres 168
13.2 POLÍTICAS DE SEGURIDAD 169
13.2.1 Dominios privados 169
13.2.2 Dominios públicos 170
13.2.2.1 Un único servidor público 170
13.2.2.2 Buzones para ampliar las capacidades de los usuarios 170
13.2.3 Dominios asociados a clases 171
13.2.4 Dominios arbitrarios 172
13.2.5 Políticas asociadas a usuarios 173
13.2.6 Dominios gestionados por un gestor de seguridad 173
13.2.7 Ejemplo de política con revocación 174
13.2.8 Ejemplo de política con control de propagación 175

14 ESPECIFICACIÓN DE LA MÁQUINA ABSTRACTA SOBRE LA QUE SE IMPLANTA EL MECANISMO BÁSICO DE PROTECCIÓN 177

14.1 EL SISTEMA INTEGRAL ORIENTADO A OBJETOS OVIEDO3 177
14.1.1 Esquema general del sistema Oviedo3 177
14.2 LA MÁQUINA ABSTRACTA CARBAYONIA 178
14.2.1 Estructura de la máquina 178
14.2.1.1 Área de Clases 179
14.2.1.2 Área de Instancias 179
14.2.1.3 Área de Referencias 180
14.2.1.4 Referencias del Sistema 180
14.2.2 Reflectividad de la máquina como aspecto destacable 180
14.3 EL LENGUAJE CARBAYÓN 180
14.3.1 Instrucciones declarativas: Descripción de clases 180
14.3.1.1 Class (clase) 181
14.3.1.2 Sección Isa (herencia) 181

14.3.1.3 Sección Aggregation (agregaciones) 181
14.3.1.4 Sección Association (asociaciones) 181
14.3.1.5 Sección Methods (declaración de métodos de la clase) 181
14.3.2 Características de las clases Carbayonia 182
14.3.2.1 Herencia virtual 182
14.3.2.2 Herencia múltiple. Calificación de métodos 182
14.3.2.3 Uso exclusivo de métodos 182
14.3.2.4 Uso exclusivo de enlace dinámico (sólo métodos virtuales) 182
14.3.2.5 Ámbito único de los métodos 182
14.3.2.6 Inexistencia de constructores y destructores 183
14.3.3 Redefinición de métodos 183
14.3.4 Definición de métodos 183
14.3.4.1 Cabecera de método 183
14.3.4.2 Refs (Referencias) 183
14.3.4.3 Instances (Instancias) 183
14.3.5 Code. Cuerpo del método 184
14.3.5.1 Instrucciones para la creación de instancias 184
14.3.5.2 Asignación de referencias 184
14.3.5.3 Invocación de métodos 185
14.3.5.4 Eliminación de instancias 185
14.3.5.5 Control de Flujo 185
14.3.5.6 Manejo de Excepciones 186
14.3.6 Jerarquía de Clases Primitivas 187

15 DISEÑO E IMPLEMENTACIÓN DEL PROTOTIPO INICIAL DE LA MÁQUINA ABSTRACTA 189

15.1 INTRODUCCIÓN 189
15.2 DISEÑO ORIENTADO A OBJETOS 191
15.3 IMPLEMENTACIÓN PRIMITIVA DE ELEMENTOS BÁSICOS 192
15.3.1 Clases primitivas y clases de usuario 192
15.4 REPRESENTACIÓN DE LOS ELEMENTOS DE LA MÁQUINA 193
15.4.1 Representación de las clases y del área de clases 193
15.4.2 Representación de los métodos 194
15.4.3 Representación de las instancias y del área de instancias 194
15.4.4 Representación de las referencias 195
15.5 FUNCIONAMIENTO GENERAL DE LA MÁQUINA 196
15.6 CREACIÓN DE ELEMENTOS 197
15.6.1 Creación de una clase de usuario 197
15.6.2 Creación de una instancia de usuario 198
15.7 EJECUCIÓN DE INSTRUCCIONES 199
15.7.1 Ejecución de una instrucción 199
15.7.2 Instrucción New <ref> 200
15.7.3 Instrucción Delete 201
15.7.4 Instrucción Assign <ref-dest>, <ref-orig> 202
15.7.5 Instrucción Call: <ref>.<método>([<params>]) 203

16 IMPLANTACIÓN DEL MECANISMO DE PROTECCIÓN EN LA MÁQUINA ABSTRACTA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS 205

16.1 INTEGRACIÓN DE CAPACIDADES CON REFERENCIAS 205
16.1.1 Permisos de longitud variable 205
16.1.2 Permiso de salto de protección 206
16.2 IMPLEMENTACIÓN DE LA INSTRUCCIÓN DE RESTRICCIÓN DE PERMISOS: FORBIDEXEC 207
16.2.1 Creación del vector de permisos 209
16.3 COMPROBACIÓN DE PERMISOS: MODIFICACIÓN DE LA INVOCACIÓN DE MÉTODOS (INSTRUCCIÓN CALL). 210
16.4 MODIFICACIÓN DE LA INSTRUCCIÓN ASSIGN 212
16.5 MODIFICACIÓN DE LA INSTRUCCIÓN NEW 213
16.6 MODIFICACIÓN DE LA INSTRUCCIÓN DELETE 214

17 ANÁLISIS DEL RENDIMIENTO DEL SISTEMA CON LA INCORPORACIÓN DEL MECANISMO DE PROTECCIÓN 215

17.1 CREACIÓN DE UNA CLASE PRIMITIVA RELOJ PARA EVALUAR EL RENDIMIENTO 216
17.2 PRUEBAS DE RENDIMIENTO 216

17.2.1 Resultados *218*
17.2.1.1 Sobrecarga del salto de protección 219
17.2.1.2 Sobrecarga de la comprobación de permisos 219
17.3 CONSIDERACIONES SOBRE EL RENDIMIENTO 220
17.3.1 La protección siempre implica una sobrecarga *220*
17.3.2 Optimizaciones en la implementación *220*
17.3.3 Número reducido de llamadas protegidas necesario en aplicaciones reales *221*
17.3.4 Complejidad de los métodos protegidos en aplicaciones reales *222*
17.3.5 Técnicas de análisis estático *222*
17.3.6 Comparaciones de rendimiento con la arquitectura de seguridad de Java *222*
17.3.7 Coste real y aceptación por los usuarios *222*

18 ELABORACIÓN DE UN ENTORNO OPERATIVO SOBRE LA MÁQUINA ABSTRACTA 225

18.1 INTRODUCCIÓN 225
18.2 SERVICIO DE SEGURIDAD 225
18.3 ENTORNO OPERATIVO 226
18.4 REQUISITOS DE DISEÑO DEL ENTORNO OPERATIVO 226
18.5 DEFINICIÓN DE UN ENLACE DEL USUARIO CON LA MÁQUINA ABSTRACTA. CLASE SYSTEM 227
18.6 OTRAS MODIFICACIONES 228
18.7 CARACTERÍSTICAS GENERALES DEL ENTORNO OPERATIVO 229
18.8 ÓRDENES DEL SHELL 230
18.8.1 Órdenes internas *230*
LDCLASS <clase.obj> 230
ULCLASS <clase> 230
LSCLASS 230
HALT 231
GRBCOL 231
EXIT 231
18.8.2 Gestión de Usuarios *231*
ADDUSR <nombre> 231
DELUSR <nombre> 231
LISTUSR 231
WHOAMI 231
18.8.3 Gestión de slots y directorios *232*
DIR / DIR@P 232
NEW [@]p<nombre> <clase> 232
REN [@p] <nombre_antiguo> <nombre_nuevo> 232
DEL [@p] <nombre> 232
MKPUB <nombre> 232
VIEW [@p] <nombre> 232
FEXEC [@p] <nombre> <método> 233
EXEC [@p] <nombre> <método> 233
18.9 DISEÑO E IMPLEMENTACIÓN DEL ENTORNO OPERATIVO 234
18.10 DISEÑO DETALLADO 235
18.11 DOCUMENTACIÓN DEL ENTORNO 237
TEOOO 237
TUser 238
TDirectory 238
TSlot 239
TShell 239

19 APLICACIÓN A OTROS SISTEMAS Y TRABAJO RELACIONADO 241

19.1 APLICACIÓN A LA PLATAFORMA JAVA 241
19.2 TRABAJO RELACIONADO 242
19.2.1 Capacidades ocultas *242*
19.2.2 El proyecto SLK y el J-Kernel *242*
19.2.3 Problemas de la implementación mediante capas añadidas *243*
19.2.4 Flexibilidad de las capacidades orientadas a objetos *243*

20 CONCLUSIONES 245

20.1 CAPACIDADES ORIENTADAS A OBJETOS 245
20.2 ALGUNOS RESULTADOS DESTACABLES 247
20.2.1 Modelo de capacidades el más adecuado para un SIOO *247*

20.2.1.1 Capacidades más adecuadas que listas de control de acceso........ 247
20.2.1.2 Peligro de las listas de control de acceso........ 247
20.2.2 Modelo propuesto de capacidades orientadas a objetos: ventajas adicionales........ 247
20.2.2.1 Cumplimiento de todos los requisitos de seguridad en el diseño de un mecanismo de protección........ 247
20.2.2.2 Uniformidad........ 247
20.2.2.3 Homogeneidad........ 248
20.2.2.4 Flexibilidad........ 248
20.2.2.5 Adaptabilidad........ 248
20.2.2.6 Sencillez de implementación y conceptual (aumento de la productividad)........ 248
20.2.2.7 Rendimiento adecuado de la protección basada en capacidades........ 248
20.2.2.8 Permisos de longitud variable. Semántica de objetos completa........ 248
20.2.2.9 Protección automática de capacidades........ 249
20.2.3 Aplicación a otros sistemas........ 249
20.2.3.1 Insuficiencia de los mecanismos de protección de la plataforma Java........ 249
20.2.3.2 Mejora de la plataforma Java........ 249
20.2.4 Implementación de prototipos........ 249
20.2.4.1 Flexibilidad del concepto de SIOO........ 249
20.2.4.2 Importancia de la reflectividad para la extensibilidad........ 249
20.3 TRABAJO Y LÍNEAS DE INVESTIGACIÓN FUTURAS........ 250
20.3.1 Estudio de patrones de comportamiento de aplicaciones........ 250
20.3.2 Desarrollo completo de un entorno de usuario........ 250
20.3.3 Mejora en la implementación de la máquina........ 250
20.3.4 Reflectividad: modelo reflectivo para soporte de monitores de referencia........ 250
20.3.5 Aplicación a la plataforma Java........ 251

BIBLIOGRAFÍA........ 253

1 Objetivos y Organización de esta Tesis Doctoral

En este capítulo se describen brevemente los aspectos básicos de que consta esta tesis, indicando los principales objetivos y contenidos de la misma. Asimismo, se enumeran las fases de su desarrollo y se describe la organización de la memoria en función de dichas fases.

1.1 Objetivos

En esta tesis se trata de desarrollar el núcleo básico de seguridad para un Sistema Integral Orientado a Objetos (SIOO), compuesto por una máquina abstracta reflectiva orientada a objetos sobre la que se establece un sistema operativo orientado a objetos. El núcleo se plasma en un mecanismo de protección[1] que se integre de manera uniforme en el modelo de objetos del sistema y cumpla los requisitos de diseño generales para un sistema de seguridad [SS75] y los específicos para un SIOO.

La protección y la seguridad son elementos cada vez más importantes en plataformas actuales como los SIOO, en las que múltiples entidades conviven de manera distribuida en el sistema e interactúan entre sí para llevar a cabo un trabajo cooperativo, y, en muchos casos, procedentes de máquinas remotas de donde se descargan bajo demanda. Es necesaria la existencia de un mecanismo de protección que posibilite la cooperación segura entre estas entidades.

Los sistemas existentes en estos momentos carecen de la uniformidad necesaria y no son adecuados para Sistemas Integrales Orientados a Objetos. Las soluciones aportadas no son lo suficientemente adecuadas como para resolver los problemas de seguridad que aparecen con el uso de la interoperabilidad de objetos distribuidos.

El mecanismo de protección diseñado permite, de forma flexible, la implementación de diferentes políticas de seguridad, así como la adaptación del mecanismo a diferentes entornos de funcionamiento del sistema. Por otro lado, mantiene totalmente la filosofía y transparencia del SIOO (en cuanto a movilidad de objetos, por ejemplo). Además, extiende la protección a un nivel de granularidad más fina que los sistemas convencionales: protege métodos individuales de cada objeto del sistema, con independencia de su tamaño o tipo, aportando más ventajas que los sistemas existentes. Por otro lado, la sobrecarga que introduce la protección es muy reducida en ambientes de trabajo realistas, lo que hace que no existan inconvenientes para la adopción del mecanismo de protección por parte de los usuarios.

[1] El mecanismo de protección, que tiene que formar parte de cualquier sistema de seguridad de un sistema operativo, se refiere a la parte del sistema que realiza el control último de acceso a los recursos (objetos). El término seguridad es más amplio y abarca en general decisiones políticas de a quién y bajo qué condiciones se deja el acceso, y se construye a partir del mecanismo de protección.

Los pilares básicos que se pretende que estén presentes en el diseño del núcleo de seguridad son tres:

- **Uniformidad.** La idea de conseguir un mecanismo de protección uniforme, de forma que todos los problemas de seguridad se resuelvan con un mecanismo único constituye un objetivo fundamental.
- **Flexibilidad.** La idea de establecer un mecanismo básico de protección capaz de dar soporte a diferentes políticas de seguridad en función de las necesidades es otro de los pilares sobre los que se apoya su diseño.
- **Integración en el modelo.** La idea de integrar el mecanismo de protección en el modelo de objetos de forma que éste constituya parte del modelo sin que ello afecte a la semántica del mismo, es otro de los objetivos a conseguir.

Además de estos objetivos, como objetivo final se busca la evaluación del modelo desarrollado mediante la implantación y valoración de los resultados obtenidos.

1.2 El proyecto de investigación en el que está enmarcada

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 seguridad constituye uno de estos aspectos y es el motivo de esta investigación.

1.3 Fases de desarrollo

En este apartado se describen las fases de trabajo en las que se divide el desarrollo de la tesis. En él no se hace mención al contenido desarrollado en cada fase, sino a los aspectos funcionales de cada una.

1. En primer lugar, se trata de estudiar la problemática de la seguridad en los sistemas de computación, qué soluciones se están aplicando al problema de la seguridad en estos momentos, en el ámbito de los sistemas existentes, tanto en entornos comerciales, como de investigación. En este estudio se conseguirá un conocimiento profundo de los modos y maneras de abordar el problema de la seguridad, y una descripción sistematizada de los diferentes modelos de seguridad existentes, y su aplicación a los sistemas reales
2. En segundo lugar, se realiza un análisis de los requisitos deseables para el diseño de un sistema de seguridad de un Sistema Integral Orientado a Objetos. Aspectos generales de seguridad y aspectos relacionados con la arquitectura del sistema integral tendrán mucho que ver con la elección de los requisitos de partida.
3. Una vez acotados los requisitos se realiza un estudio crítico sobre los mecanismos de seguridad existentes en algunos sistemas relevantes, desde la perspectiva que imponen estos requisitos.
4. Como consecuencia de los requisitos impuestos, y los conocimientos adquiridos en la primera y tercera fase, se elabora el núcleo de seguridad adecuado que puede regir el comportamiento de un sistema integral. Esta fase se concibe como resultado del proceso de crítica, reflexión y selección de posibilidades iniciado tras la primera fase.

A ello hay que añadir un proceso de síntesis y consecución de un nuevo modelo de seguridad, que se ajusta mejor a los requisitos de partida y que produce mayores ventajas que los modelos existentes. Finalmente, dentro de esta fase, se formalizará este modelo, analizando sistemáticamente todos los aspectos que intervienen en él.

5. Una vez formalizado el modelo, se procede a la descripción de sus virtudes, que dan crédito a la bondad de éste y mayor adecuación respecto a otros ya existentes. Además de ello, se apoya la demostración en una implantación real sobre un sistema integral, el sistema Oviedo3, con la máquina abstracta Carbayonia, y se realizan mediciones que justifican la bondad del modelo.

FASE I Estado del Arte	FASE II Análisis de Requisitos	FASE III Estudio de sistemas relevantes	FASE IV Elaboración del núcleo de seguridad	FASE V Demostración de la idea

Figura 1.1. Fases de desarrollo

1.4 Organización de la memoria

El documento que aquí se presenta está organizado según las fases de desarrollo descritas anteriormente.

El Capítulo 1: “Objetivos y Organización de esta Tesis Doctoral”, es éste que aquí se presenta.

El Capítulo 2: “Introducción”, pretende justificar la necesidad de investigación existente en el ámbito de la seguridad en un sistema integral orientado a objetos. Para ello se describen los problemas de interoperabilidad y se justifica la inclinación hacia un sistema integral orientado a objetos en los entornos de computación distribuidos que están siendo utilizados. A continuación se describen los aspectos relativos a la seguridad en estos entornos y se plantea la necesidad de un nuevo modelo, apoyándose en el hecho de que los modelos existentes tienen bastantes limitaciones.

El Capítulo 3: “Conceptos de Seguridad”, está dedicado a la definición de los conceptos y aspectos que se barajan en el campo de la seguridad informática. En él se especifica el sentido que se le va a dar a los vocablos seguridad y protección, se describen los aspectos a tener en cuenta en la seguridad de un sistema de computación, se analizan los tipos de protección que ofrecen los sistemas, se describen los principios de diseño y los principales problemas que se deben resolver. En definitiva, se trata de delimitar el ámbito de trabajo en el que se mueve la tesis doctoral y fijar la terminología utilizada.

El capítulo 4: ”Modelos de Seguridad”, describe los principales modelos de seguridad que han sido formalizados hasta el momento, y de los que se derivan las diversas implementaciones existentes.

El capítulo 5: “Resumen de la Arquitectura de un Sistema Integral Orientado a Objetos”, describe las principales características de un Sistema Integral Orientado a Objetos basado en una máquina abstracta orientada a objetos, sistema sobre el cual se debe desarrollar el mecanismo de protección. En él se desarrollan los aspectos que definen el modelo único de objetos y la arquitectura de referencia de la máquina abstracta reflectiva orientada a objetos que es el substrato del sistema, sobre el que se construye un sistema operativo orientado a objetos. Además, se muestran las diferentes alternativas disponibles para extender la funcionalidad del sistema.

El capítulo 6: "Requisitos de un Mecanismo Básico de Protección para un Sistema Integral Orientado a Objetos", constituye una descripción de los requisitos que se piden a un mecanismo de protección dentro del contexto de los sistemas integrales orientados a objeto y teniendo en cuenta las características de la plataforma de integración del mecanismo.

El capítulo 7: "Panorama de Seguridad en algunos Sistemas Operativos", se dedica a realizar una visión general del estado del arte en cuanto a seguridad se refiere en los sistemas de computación. Para ello se han elegido un conjunto de sistemas que representan las distintas filosofías de diseño de sistemas operativos, en cuanto al aspecto del tratamiento de la seguridad. De cada uno de estos sistemas, se lleva a cabo una pequeña descripción de sus características generales, un resumen del tratamiento que realiza de la seguridad y una crítica, de aspectos a favor y en contra, que pueden extraerse del tratamiento de seguridad diseñado. En esta crítica se intentan homogeneizar criterios basándose en los principios generales de diseño de un sistema de seguridad.

El capítulo 8: "Seguridad en la Plataforma Java", supone una continuación del capítulo anterior. Se ha dedicado al tratamiento de la seguridad en la plataforma Java, por la importancia que esta plataforma tiene en los sistemas de computación e Internet.

Estos ocho primeros capítulos pueden considerarse el primero de los tres bloques que constituyen esta tesis doctoral. Su contenido abarca los aspectos teóricos y de conocimiento que inducen a la elaboración del verdadero diseño del núcleo de seguridad deseado. Los siguientes capítulos, del 9 al 13 inclusive, delimitan los aspectos que intervienen en la elaboración del mecanismo de protección propiamente dicho.

El capítulo 9: "Elección de Capacidades como Mecanismo Base de Protección", constituye una disertación acerca de cuáles son las razones por las que se elige el modelo de capacidades como modelo básico de protección del sistema. Para ello, se justifican los inconvenientes de otros modelos, que hacen que sean rechazables, y se apoya la decisión demostrando que ésta puede solucionar los problemas de seguridad comúnmente detectados.

En el capítulo 10: "Estudio de Capacidades", se realiza un estudio detallado de todas las características del modelo de capacidades. Dado que ha resultado elegido, resulta interesante conocerlo con detenimiento. Aunque ya se había hablado de él en el capítulo de modelos, sin embargo, aquí se complementa su descripción con aspectos no tratados, como posibles implementaciones, diseño de los permisos, etc.

El capítulo 11: "Diseño del Modelo de Protección", desarrolla el diseño ideado para la implantación del mecanismo. El modo y manera en que se integren las capacidades en un núcleo de protección para un sistema integral orientado a objetos, así como el diseño detallado que se propone del mecanismo serán objeto de descripción de este capítulo. En esta descripción se justifica adecuadamente cada decisión tomada, siendo la más importante la decisión de integrar las capacidades con las referencias de la máquina abstracta. El resultado obtenido es la idea de un mecanismo nuevo, que no ha sido usado con anterioridad y que aporta ventajas adicionales a los sistemas existentes. Su sencillez y la capacidad de éste para resolver de forma elegante los problemas fundamentales de los sistemas de seguridad, constituyen su mejor tarjeta de presentación.

Es en el capítulo 12: "Ventajas del Modelo de Protección Diseñado", donde se justifican las ventajas que aporta el mecanismo ideado. De nuevo se usan los principios básicos de diseño como medida de la bondad del método, además de los requisitos propios de un SIOO.

El capítulo 13: "Políticas de Seguridad sobre el Mecanismo Básico", describe a modo de ejemplo cómo diferentes políticas de seguridad pueden implementarse sobre el mecanismo de protección ideado. El aspecto del diseño de políticas, si bien no es objeto de esta tesis

doctoral, si es tratado en ella. La descripción de distintos tipos de políticas utilizables sobre el núcleo de seguridad, constituye una demostración de la flexibilidad del mecanismo, y justifican la bondad de su diseño. No obstante, se incluye en el bloque principal de la tesis puesto que su elaboración supuso también la inclusión de nuevas ideas de diseño, que si bien son colaterales, complementan el desarrollo del núcleo de seguridad.

Los restantes capítulos de esta memoria, del 14 al 19, constituyen el tercer bloque de división de las tesis doctoral. En ella se pretende demostrar de forma práctica las tesis sostenidas. Para ello se lleva el diseño a una plataforma de desarrollo donde se incluye su prueba y valoración. La plataforma Oviedo3 es la elegida para este desarrollo, y los niveles correspondientes a su máquina abstracta y su sistema operativo se ven alterados para incluir los aspectos relativos a la seguridad. En estos capítulos se describen los aspectos de implantación que se han llevado a cabo a tal efecto.

El capítulo 14: "Especificación de la Máquina Abstracta sobre la que se Implanta el Mecanismo Básico de Protección", describe la plataforma de implantación Oviedo3, sus características generales, y la descripción de la máquina abstracta y el lenguaje que sirve de base al sistema.

El capítulo 15: "Diseño e Implementación del Prototipo Inicial de la Máquina Abstracta", describe en términos generales el diseño e implementación de un prototipo inicial de la máquina en la que aún no existe el mecanismo de protección.

El capítulo 16: "Implantación del Mecanismo de Protección en la Máquina Abstracta del Sistema Integral Orientado a Objetos", describe la implantación del mecanismo de protección del sistema a partir del prototipo descrito anteriormente, con lo que ello conlleva en cuanto a modificación del diseño y la implementación de la máquina.

El capítulo 17: "Análisis del Rendimiento del Sistema con la Incorporación del Mecanismo de Protección", constituye un análisis del rendimiento de la implementación realizada incluyendo el mecanismo de protección. En él se describen los resultados de obtenidos tras realizar una batería de pruebas con el fin de verificar el mecanismo y medir tiempos de sobrecarga que éste añade. Se encuentra que en un entorno realista la sobrecarga es mínima y no supondría un inconveniente para la adopción del mecanismo.

El capítulo 18: "Elaboración de un Entorno Operativo sobre la Máquina Abstracta", describe el diseño e implementación de un entorno operativo simple como ejemplo práctico que muestre las posibilidades del sistema y el mecanismo de protección en conjunto. Como parte del entorno operativo se incluye una política de seguridad sencilla que utiliza el mecanismo de protección como base subyacente, demostrando así la flexibilidad de éste. Además de ello, el entorno permite manipular las capacidades de manera más cómoda en el nivel de interfaz de usuario con un intérprete de línea de órdenes.

El capítulo 19: "Aplicación a otros Sistemas y Trabajo Relacionado", describe cómo el modelo ideado en este trabajo puede adaptarse a otros sistemas como Java. También se introducen algunos trabajos relacionados en el área de introducción de la protección basada en capacidades a entornos ya existentes, mostrando cómo el mecanismo de protección propuesto puede sustituir y/o complementar ventajosamente estos trabajos.

El capítulo 20: "Conclusiones", se resumen las principales conclusiones obtenidas de la elaboración del trabajo y se perfilan algunas líneas de investigación futuras que marcan el camino a seguir en la investigación empezada con esta tesis.

Por último aparece la bibliografía empleada en el trabajo.

2 Introducción

En este capítulo se pretende justificar la necesidad de investigación existente en el ámbito de la seguridad en un Sistema Integral Orientado a Objetos. Para ello se describen los problemas de interoperabilidad y se justifica la inclinación hacia un sistema integral orientado a objetos en los entornos de computación distribuidos que están siendo utilizados. A continuación se describen los aspectos relativos a la seguridad en estos entornos y se plantea la necesidad de un nuevo modelo, apoyándose en el hecho de que los modelos existentes tienen bastantes limitaciones.

2.1 Necesidad de un sistema integral orientado a objetos

En los últimos años se ha producido una notable tendencia hacia sistemas distribuidos heterogéneos, conectados entre sí y compartiendo información. La expansión de la red Internet y el uso de Java son ejemplos claros de ello. Con este tipo de sistemas, se extiende el uso de arquitecturas cliente/servidor donde los servicios se reparten por distintas máquinas de la red.

Por otra parte, las tecnologías orientadas a objetos se están implantando en mayor o menor medida en todos los ámbitos de la informática. El uso masivo de esta tecnología ha producido la necesidad de integrar nuevas herramientas que ayuden a su desarrollo y explotación. Así por ejemplo el estándar CORBA [OMG95] describe la arquitectura que debe tener un gestor de peticiones entre objetos que permita la interoperabilidad de los mismos. La utilización de los *applets* de Java, el desarrollo de sistemas de componentes y la aparición de los agentes, vienen a corroborar esta tendencia hacia la distribución masiva y el empleo de tecnologías orientadas a objetos para resolver los problemas que esto presenta.

2.1.1 El problema de la desadaptación de impedancias

Sin embargo, la incorporación del paradigma de la orientación a objetos (OO) no se está llevando a cabo de una manera integral en todos los componentes de un sistema, sino que se aplica de forma parcial sólo en alguno de ellos. Ello implica un grave problema de desadaptación de impedancias o salto semántico entre diferentes elementos del sistema. Este problema se produce cuando un elemento (por ejemplo una aplicación OO) debe interactuar con otro (por ejemplo un sistema operativo).

Abstracciones inadecuadas de los sistemas operativos

El hardware convencional sobre el que deben funcionar las tecnologías OO sigue aún basado en versiones evolucionadas de la arquitectura de Von Neumann. Los sistemas operativos ofrecen abstracciones basadas en este tipo de hardware, más adecuadas para el paradigma procedimental estructurado.

Es muy común la utilización de lenguajes, sistemas de gestión de bases de datos orientados a objetos sobre sistemas operativos tradicionales, interfaces gráficas de usuario convencionales en las que funcionan aplicaciones cuya interfaz es orientada a objetos, etc.

Esto provoca un grave problema de desadaptación de impedancias, al tener que estar cambiando constantemente de paradigmas en función del elemento del sistema con que se

esté tratando. Así, por ejemplo, un programa escrito en un lenguaje OO cuando necesite utilizar un servicio del sistema operativo deberá utilizar el paradigma del sistema operativo (normalmente de tipo procedimental) en lugar del paradigma OO que se usa en el resto de la aplicación.

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

Otro problema 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. El compilador no tiene ningún mecanismo que permita realizar la invocación de métodos en espacio 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. 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.

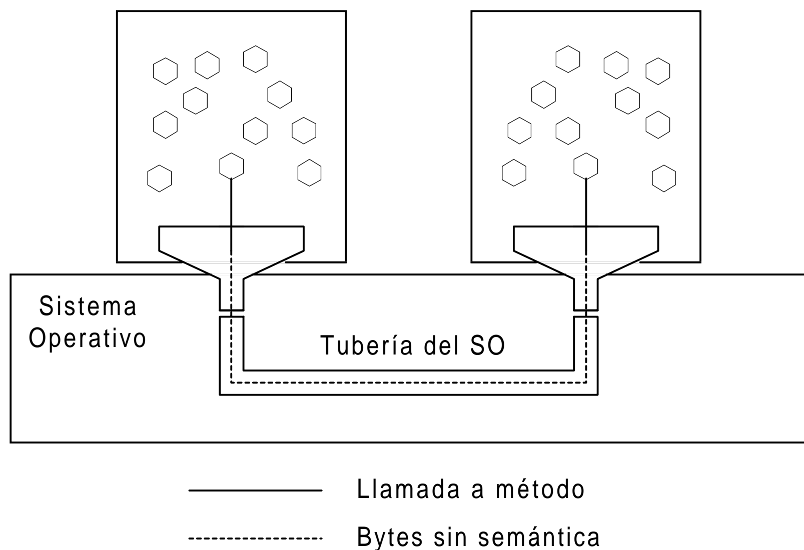


Figura 2.1: Comunicación entre objetos de espacios de direcciones diferentes

2.1.1.2 Desadaptación de interfaces

Otro problema se produce cuando las aplicaciones OO necesitan utilizar los servicios del sistema operativo. La interfaz de utilización está enfocada al paradigma procedimental, normalmente en forma de una llamada al sistema (llamada a procedimiento). El resultado es que el programador/usuario del sistema se ve obligado a utilizar dos paradigmas diferentes en el desarrollo de aplicaciones.

2.1.2 El problema de la interoperabilidad entre modelos de objetos

Es habitual también que entre diferentes elementos del sistema que usen el paradigma de la orientación a objetos se produzca desadaptación. 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. Las propiedades de los objetos de cada sistema pueden diferir, por ejemplo un modelo puede tener constructores y otros no, etc.

Por ejemplo, en una aplicación desarrollada en el lenguaje C++, con el modelo de objetos C++ 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.

Para solucionar el problema de la interoperabilidad se recurre a la introducción de capas software de adaptación. CORBA es un ejemplo de este tipo de software. CORBA define un modelo de objetos propio en el que sólo se especifica la interfaz de un objeto. Para cada lenguaje se define una correspondencia entre el modelo de objetos propio y su interfaz dentro del modelo CORBA. Una vez dada esta correspondencia ya se tiene información suficiente para generar el software de adaptación requerido. Éste se incorpora a la implementación de los objetos cliente y servidor. Los intermediarios de peticiones de objetos (ORB, *Object Request Broker)* en las máquinas cliente y servidora junto con este software incluido en los programas cliente y servidor permiten la invocación de métodos.

2.1.3 Problemas de las capas de adaptación sobre sistemas tradicionales

La adición de capas para salvar los problemas de desadaptación que presenta el paradigma de la orientación a objetos provoca una serie de inconvenientes:

- **Disminución en el rendimiento global del sistema**. A causa de la sobrecarga debida a la necesidad de atravesar todas estas capas software para adaptar los paradigmas.
- **Falta de uniformidad y transparencia**. Estas soluciones no son todo lo transparentes 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.
- **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. Además estos programas sólo pueden funcionar en máquinas a las que también se haya portado los sistemas de adaptación que se utilicen.
- **Aumento de la complejidad del sistema**. Al existir tantas capas el sistema se vuelve más complejo, y por tanto más difícil de entender y de analizar en su funcionamiento.

2.2 Sistemas Integrales Orientados a Objetos

Esta situación planteada, necesita nuevas aproximaciones de investigación que permitan solucionar el problema, como son los **sistemas integrales orientados a objetos** (SIOO) que den soporte directo y utilicen exclusivamente el paradigma de la orientación 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.

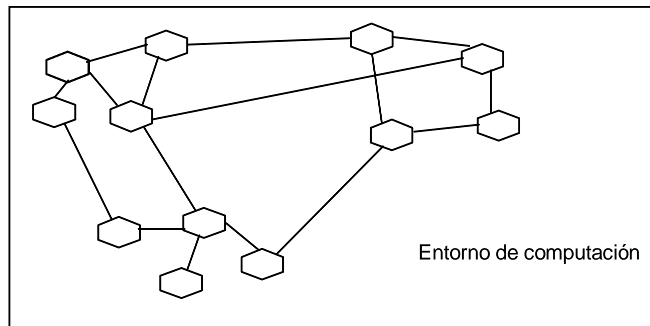


Figura 2.2: Entorno de computación compuesto por un conjunto de objetos homogéneos

2.2.1 Requisitos de un sistema integral orientado a objetos

Los requisitos que deben estar presentes en un sistema de este estilo [Álv98] se resumen brevemente a continuación.

2.2.1.1 Uniformidad conceptual en torno a la orientación a objetos

La idea fundamental en un sistema de este estilo es proporcionar uniformidad conceptual en torno a la orientación a objetos, que conduce a los puntos siguientes:

- **Modo de trabajo exclusivamente orientado a objetos**. La única abstracción es el objeto, de cualquier granularidad, que encapsula toda la semántica. El objeto es una entidad activa cuyo único comportamiento es crear nuevas clases que hereden de otras, crear objetos de una clase, y enviar mensajes a otros objetos.
- **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

2.2.1.2 Transparencia de distribución y persistencia

La posibilidad de que un objeto se mueva a un nodo diferente del sistema y que un objeto sea persistente debe ser transparente para el usuario.

2.2.1.3 Heterogeneidad y portabilidad

El sistema debe poder funcionar en plataformas hardware heterogéneas, y además debe ser sencillo portarlo a las mismas.

2.2.1.4 Seguridad

Debe existir un mecanismo de protección en el sistema que permita que el entorno de computación sea seguro frente a ataques malintencionados o errores lógicos. Este es el aspecto del que se ocupa este trabajo con detalle.

2.2.1.5 Concurrencia

El sistema tendrá un modelo de concurrencia que permita la utilización de los objetos aprovechando el paralelismo.

2.2.1.6 Multilenguaje/Interoperabilidad

El sistema podrá ser utilizado mediante distintos lenguajes y los objetos creados con estos podrán interoperar entre sí.

2.2.1.7 Flexibilidad

Debe poderse añadir, eliminar, o modificar la funcionalidad del sistema de manera sencilla.

2.2.2 Arquitectura de un Sistema Integral Orientado a Objetos

Una arquitectura prometedora para construir un sistema integral orientado a objetos logrando las propiedades anteriores se basa en la utilización de una máquina abstracta reflectiva orientada a objetos. Esta máquina constituye el substrato del sistema, y proporciona el modelo de objetos básico y el soporte de objetos.

Sobre ella se desarrolla un sistema operativo orientado a objetos constituido por un conjunto de objetos normales que proporcionan las funcionalidades tradicionales de sistema operativo, como control de concurrencia, persistencia, distribución, etc.

Sobre esta base fundamental del sistema se disponen el resto de los elementos del entorno: sistemas de gestión de bases de datos, subsistemas gráfico y multimedia, lenguajes y compiladores orientados a objetos, etc.

En un capítulo posterior se describirá con más detalle la arquitectura propuesta para un sistema integral orientado a objetos.

2.3 Seguridad de un sistema integral orientado a objetos

Existen múltiples aspectos a tratar en un sistema integral orientado a objetos, todos los cuales son sujeto de activa investigación actualmente. Aspectos como la concurrencia, la distribución, la persistencia de objetos o la seguridad cobran especial relevancia en este contexto. En esta tesis doctoral la investigación se centra en la búsqueda de un **mecanismo de protección** (**mecanismo de control de acceso**) que resulte óptimo para el objetivo global del sistema integral en su conjunto.

Dado que en un SIOO todo lo existente son objetos que se comunican entre sí, el sistema de seguridad1 que se proporcione deberá proteger el acceso a operaciones entre objetos siempre que así se desee. Todos los objetos del sistema tienen la misma categoría, por lo que el mecanismo debe tratar la protección de manera uniforme, sin distinguir diferentes mecanismos en función del tipo de objetos con los que se trate. Así pues, tanto los objetos del sistema operativo como objetos normales de aplicaciones del usuario deben tener el mismo mecanismo básico de protección.

La búsqueda de un mecanismo de protección adecuado y el diseño de un núcleo de seguridad que proporcione protección a los objetos de un sistema integral orientado a objetos, es por tanto, el reto que en esta tesis se propone, y que se irá desarrollando en este documento.

2.4 Algunas soluciones planteadas para el control de acceso

Con la progresiva expansión de la interoperabilidad de objetos, el problema de la seguridad se amplía no sólo a la tradicional protección de ficheros de los sistemas operativos, si no al control de la comunicación entre objetos. La necesidad de establecer soluciones para este problema se agudiza con esta expansión.

---

[1] En realidad se estará tratando con un mecanismo de protección, que tiene que formar parte de cualquier sistema de seguridad de un sistema operativo. En general, cuando se utilice el término "seguridad" se estará haciendo referencia en el sentido concreto del mecanismo de protección.

El problema fundamental se presenta cuando se desean comunicar objetos. Es necesario poder restringir el acceso de forma que sólo aquellos objetos que se consideren fiables y necesiten invocar un método de otro, puedan hacerlo. Al resto se le deberá denegar este acceso en caso de que lo solicite (por ejemplo en el caso de un objeto "Estudiante" que desee invocar el método "cambiarNota" de un objeto "Calificación").

Las soluciones propuestas siguen diferentes planteamientos. Sistemas como Choices [CM90] o Guide [Hag94] utilizan **listas de control de acceso**, asociadas a cada grupo de objetos localizados en un espacio de direcciones. En estas listas se almacenan los nombres de los objetos externos que pueden invocar métodos de objetos que residen en un espacio de direcciones dado (y el nombre de los métodos que pueden invocar). Sólo cuando se intenta invocar un método de un objeto situado en un espacio de direcciones diferente, se activa el control de acceso para permitir la operación (que implica realizar una consulta en la lista de control de acceso correspondiente).

Otros sistemas, como Amoeba [TMV86], Mungi [VRH93], KeyKOS [Lan89] e Hydra [W74] utilizan **capacidades** para el control de acceso. Las capacidades son entradas (*tickets*) que el objeto invocador debe poseer para poder acceder a una operación de un objeto e indican el objeto al que hacen referencia y las operaciones que permite invocar la capacidad. El mecanismo de protección se encarga de pedir la capacidad y comprobarla cuando se hace una invocación de método a un objeto, y comprobar que se tiene permiso para invocar el método indicado.

Algunos sistemas como Multics [CSC72], utilizan mecanismos basados en **flujo de información,** donde los objetos se dividen en categorías y su acceso depende de la clasificación y categoría. El arquetípico caso de los entornos militares donde se clasifica la información (*top secret*, confidencial, etc.) ilustra claramente esta situación.

En la actualidad, el entorno proporcionado por Internet, en el que un sistema local puede recibir a través de la red *applets* o agentes móviles que se ejecutan en el sistema, representan una grave amenaza a la seguridad si no se realiza un estricto control de acceso. Las distintas versiones de la plataforma Java [KJS96] han ido incorporando mejores sistemas de protección, que comenzaron por el método de la "caja de arena", donde las clases se dividían en fiables, y no fiables. Las fiables tenían acceso a cualquier dato del disco local y las no fiables sólo a aquellos directorios definidos en la caja de arena. Este mecanismo dio paso a otros mecanismos más completos debido a la escasa flexibilidad que proporcionaba.

2.4.1 Inconvenientes de los planteamientos adoptados

Uno de los principales inconvenientes planteados es la falta de uniformidad del mecanismo de protección. Estos sistemas en su conjunto no disponen de un único mecanismo, sino que suelen existir varios, en función de los recursos que se vayan a proteger. Así por ejemplo, la seguridad de los recursos del sistema operativo suele ser tratada de manera independiente a la seguridad de los recursos que puede manejar el usuario. Por otra parte, no es acostumbrada la existencia de un mecanismo uniforme que se aplique de igual manera para todos los objetos de usuario. En general, el espacio de direcciones en el que se ejecuta un grupo de objetos constituye la primera forma de protección. Sólo se emplean otros métodos de protección cuando el acceso sobrepasa los límites de este espacio, suponiéndose una total confianza dentro de los objetos que comparten un mismo espacio de direcciones, ya que no hay restricciones en su intercomunicación.

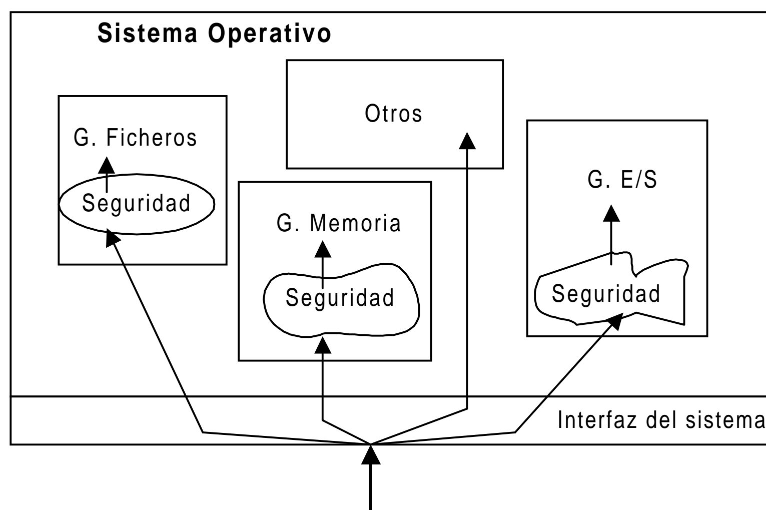


Figura 2.3: Múltiples módulos de seguridad diferentes en un sistema operativo

Por otra parte, los métodos empleados no resultan demasiado flexibles. Algunos mecanismos no contemplan la protección en el nivel de operaciones de los objetos, si no directamente en el nivel de objetos. Es decir se puede acceder a todas las operaciones de un objeto o a ninguna, sin contemplarse operaciones individuales (la granularidad de protección no es muy fina). Otros restringen la protección a un número máximo de operaciones de un objeto. En otros casos, el acceso no es discrecional, sino que depende de la clasificación a la que pertenezca.

2.4.2 Núcleo de seguridad para un sistema integral Orientado a Objetos

Las carencias encontradas en materia de seguridad en los sistemas actuales, y la falta de mecanismos integrados de forma uniforme en sistemas que empleen tecnología orientada a objetos, ha animado a este grupo de investigación a intentar desarrollar un mecanismo de protección que constituya el núcleo seguridad de un sistema integral orientado a objetos.

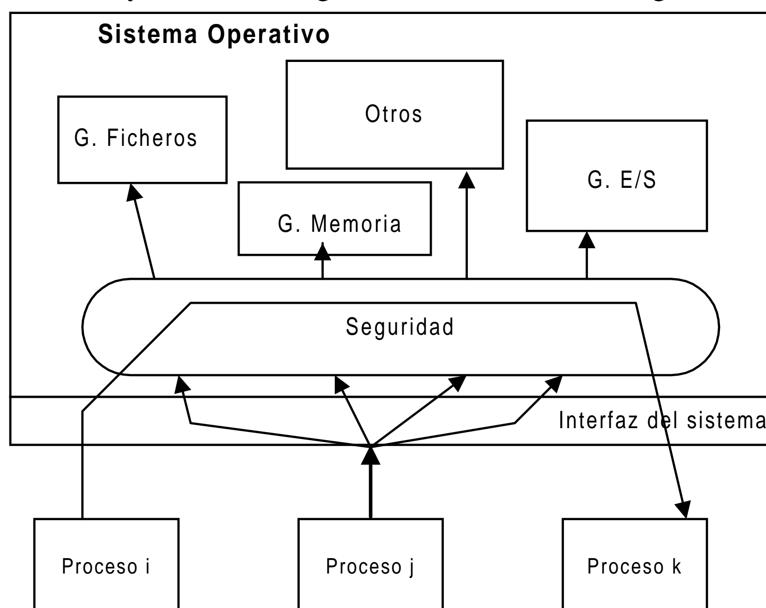


Figura 2.4: Núcleo de seguridad homogéneo para un sistema de computación

Todo el esfuerzo de esta tesis doctoral se dirige hacia este objetivo. En los capítulos sucesivos se verán de forma pormenorizada todos los aspectos que conducen a la obtención de un núcleo de seguridad, así como los requisitos impuestos y las ventajas obtenidas de él.

3 Conceptos de Seguridad

En este capítulo se definen los aspectos que engloban la seguridad de un sistema, y se acota el ámbito de desarrollo de esta tesis con respecto a la seguridad. Aquí por tanto, se establece la terminología con la que se va a trabajar en el resto del documento. Por último, se describen los principios de diseño establecidos para un buen mecanismo de seguridad y se discuten los principales problemas que debe resolver un sistema de seguridad.

3.1 Introducción

Desde los primeros sistemas informáticos diseñados, hasta los existentes en la actualidad, ha ido aumentando la cantidad y también los tipos de datos que controlan y manipulan los cada vez más potentes ordenadores.

Con anterioridad a la aparición de la informática, los datos se mantenían seguros por medios físicos como archivadores con cerradura de combinación, pero con la aparición del ordenador es necesario el uso de otras herramientas que impidan el acceso a intrusos que desean conocer la información.

Existen muchos aspectos relativos a la seguridad que deben ser contemplados en un sistema informático. Uno de ellos es el de la autenticación de los individuos que desean entran en el sistema, siendo necesario comprobar que sólo los autorizados deben poder acceder a recursos y datos que éste posee.

Una vez dentro, existen una serie de recursos, tanto hardware (dispositivos, memoria, procesador, etc.) como software (ficheros, estructuras de datos, etc.) a los que el cliente desea tener acceso. Este acceso deberá ser exclusivo en ocasiones, y compartido en otras. En cualquier caso, es necesario proteger adecuadamente los recursos, de forma que sólo los clientes autorizados puedan hacer uso de ellos.

Con la introducción de los sistemas distribuidos y el empleo de redes y servicios de comunicación para transportar datos, han sido necesarias medidas adicionales de seguridad, complicándose la protección considerablemente. Por un lado es preciso proteger los datos durante la transmisión, evitando que éstos puedan ser observados, o modificados durante su viaje. Por otro lado, la posibilidad de comunicación abre nuevos agujeros de seguridad, y será necesario evitar que programas que vienen por la red obtengan o modifiquen información confidencial localizada en la máquina cliente.

Por último, la tendencia cada vez más acusada hacia las arquitecturas cliente/servidor, donde ciertas máquinas ofrecen servicios a los que tendrán acceso los clientes amplía la necesidad de protección del puro ámbito de los recursos que ofrece el sistema operativo a los recursos ofrecidos por el propio usuario, necesitando para ello un mecanismo de protección interno que lo permita.

Así pues, la seguridad de un sistema de computación es un aspecto clave y de gran importancia, que ha aumentado en complejidad con la proliferación de los sistemas distribuidos, la red Internet y las arquitecturas cliente/servidor.

En este capítulo se va a realizar un análisis de los aspectos que integran la seguridad de un sistema, y los conceptos relativos a ésta: definición de términos, clasificación, y requisitos relacionados con un sistema de seguridad.

3.2 Aspectos de la seguridad en un sistema de computación

Aunque existen diversas clasificaciones de los aspectos que abarca la seguridad de un sistema, la más utilizada agrupa ésta en los tres apartados siguientes [Gos91].

- Seguridad en el acceso al sistema
- Seguridad en el uso de recursos y servicios
- Seguridad en el uso de redes

3.2.1 Seguridad en el acceso al sistema: Autenticación de usuarios

Sólo las personas registradas como usuarios pueden acceder a los recursos que se ofrecen en el sistema. Para ello es necesario establecer un sistema de acceso que asegure la imposibilidad de acceso de intrusos con fines malintencionados.

A este aspecto de la seguridad se le suele denominar *Autenticación de Usuarios*. El sistema operativo debe encargarse de esta función, interactuando con el usuario cuando éste inicia la sesión. La petición de una clave (contraseña) al usuario, y en ocasiones alguna información adicional, son las medidas habituales que toma el sistema operativo para la autenticación del cliente. La protección de la información que usa el sistema operativo para la autenticación es vital para evitar que clientes malintencionados descubran información que les permita hacerse pasar por otros clientes con iguales o mayores privilegios.

La posibilidad de conexión entre máquinas ha multiplicado el riesgo de accesos malintencionados. Los sistemas operativos como Unix, ampliamente utilizado, cuentan con bastantes agujeros de seguridad, que se han ido descubriendo y corrigiendo, pero que han provocado y siguen provocando problemas considerable importancia en numerosos sistemas.

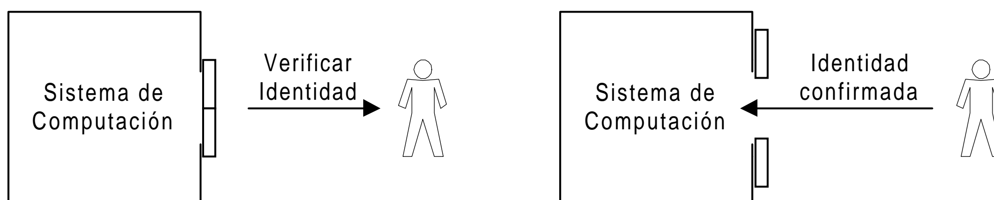


Figura 3.1: Autenticación de usuarios en un sistema de computación

3.2.2 Seguridad en el uso de recursos y servicios

El sistema operativo ofrece al usuario[1] una serie de recursos y abstracciones: memoria, ficheros, procesos, dispositivos, etc. El sistema operativo debe disponer de un mecanismo de protección que asegure que los usuarios tienen su información a salvo de otros posibles usuarios. Por tanto, deberá proteger recursos como la memoria o los ficheros.

Por otra parte, el propio usuario puede ofrecer sus servicios a otros clientes. En este caso, deberá existir también un mecanismo de protección para estos servicios. Aunque este

[1] En realidad los recursos del sistema están disponibles para los elementos activos que el sistema reconoce (abstracciones del sistema), como por ejemplo procesos en los sistemas operativos tradicionales. Es decir, el sistema no ofrece los recursos a las personas físicas, si no a las entidades internas que reconoce (que suelen tener alguna manera de ligarse a las personas físicas).

mecanismo podría ofrecerlo el propio servicio de usuario, lo más adecuado será que lo ofrezca el sistema y así sea independiente del servicio y uniforme para todos los servicios que ofrezca cualquier cliente.

El sistema operativo debe proporcionar, por tanto el control de acceso a los recursos. Se trata de controlar a qué objetos (ficheros, memoria, otros programas, etc.) puede acceder un programa y de qué manera. El acceso significa qué tipo de operaciones se pueden hacer sobre dichos objetos. Ejemplos de estas operaciones incluyen la lectura de un fichero, creación y eliminación de objetos, la comunicación con otros programas, etc.

La protección de los recursos suele ser tratada en general de forma heterogénea, según sea el recurso a proteger, dando una solución específica para cada tipo de recursos.

Más adelante se describirán con más detalle los mecanismos de control de acceso (**mecanismos de protección**) que pueden utilizarse para la gestión de los recursos.

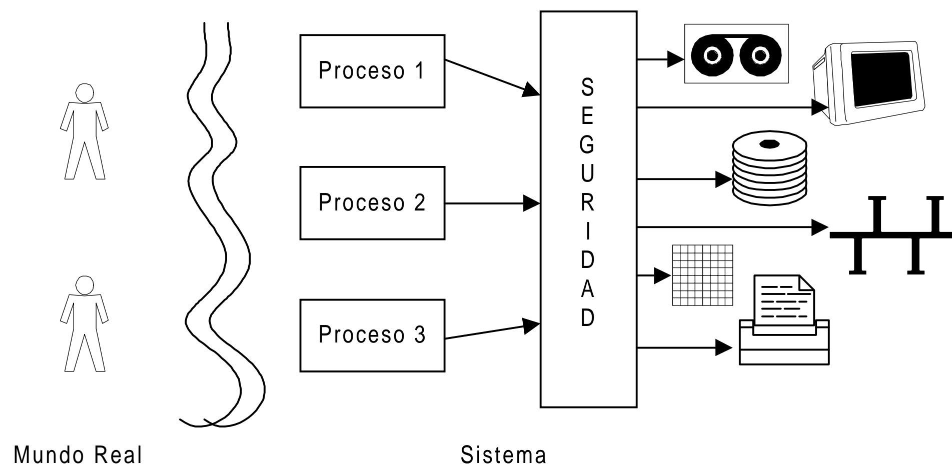


Figura 3.2: Seguridad en el uso de recursos y servicios

3.2.3 Seguridad en el uso de redes

Es necesario establecer mecanismos para evitar que se puedan producir escuchas o alteraciones de los datos de la red. Las amenazas de seguridad en las redes se clasifican en activas y pasivas. Las amenazas pasivas se presentan mediante escuchas en la transmisión de la información, a través de la revelación del contenido directamente o bien mediante el análisis del tráfico, que se emplea cuando los mensajes van encriptados (cifrados). Las amenazas pasivas no alteran el contenido de los datos y pueden prevenirse mediante el cifrado. Las activas suponen la modificación de la información que fluye por la red.

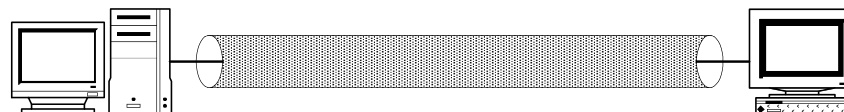


Figura 3.3: Seguridad en el uso de redes

3.3 Elementos del sistema a los que afecta la seguridad

En la seguridad de los sistemas de computación se abordan los siguientes requisitos [Sta97]:

- **Confidencialidad**. Exige que los elementos del sistema sean visibles solamente a grupos autorizados.
- **Integridad**. Exige que los elementos del sistema sólo sean modificados por los grupos autorizados.
- **Disponibilidad**. Exige que los elementos del sistema estén disponibles sólo a grupos autorizados.

Los elementos del sistema afectados por la seguridad se clasifican en: Hardware, Software, Datos, y Líneas de comunicación.

En la siguiente tabla se identifican las amenazas de seguridad que pueden perjudicar a cada requisito de seguridad y elemento del sistema:

Elemento	Confidencialidad	Integridad	Disponibilidad
Hardware			Robo o sobrecarga de equipos, eliminando el servicio
Software	Realización de copias no autorizadas del software	Alteración de un programa en funcionamiento haciéndolo fallar durante la ejecución o haciendo que realice alguna tarea no pretendida	Eliminación de programas denegando el acceso a los usuarios
Datos	Lecturas de datos no autorizadas. Revelación de datos ocultos de manera indirecta por análisis de datos estadísticos.	Modificación de archivos existentes o invención de nuevos	Eliminación de archivos, denegando el acceso a los usuarios
Líneas de Comunicación	Lectura de mensajes. Observación de la muestra de tráfico de mensajes.	Mensajes modificados, retardados, reordenados o duplicados. Invención de mensajes falsos.	Destrucción o eliminación de mensajes. Las líneas de comunicación o redes se hacen no disponibles.

Tabla 3.1: Elementos del sistema a los que afecta la seguridad

3.4 El problema del control de acceso (protección)

3.4.1 Objetivo de esta tesis doctoral

Como se ha dicho anteriormente, el control de acceso es uno de los tres grandes apartados en los que se dividen los problemas de seguridad de un sistema.

El desarrollo de esta tesis se basa precisamente en la búsqueda un buen mecanismo de control de acceso que mejore en alguna medida los que existen en la actualidad, especialmente dentro del contexto de un sistema integral orientado a objetos. Consecuentemente, a partir de este momento, todos los conceptos relacionados con seguridad

que aparezcan se centrarán en la seguridad sobre el uso de recursos y servicios, es decir, en el control de acceso (protección). Los otros dos ámbitos: autenticación de usuarios y seguridad en el uso de redes, no son motivo de discusión en ninguna de las partes de que consta esta tesis.

3.4.2 Aspectos que abarca el control de acceso

Cuando se trata el problema del control de acceso, existen cuatro aspectos que deben ser tratados: prevención, limitación, concesión y revocación [Shap99]

3.4.2.1 Prevención del acceso

Es necesario asegurar que no es posible que una persona tenga acceso a información privada de la otra. Así por ejemplo, habrá que evitar que alguien pueda leer o borrar información privada, sobre la que no debe tener acceso. Por ejemplo, que otro usuario pueda borrar documentos propios privados.

Este aspecto de la seguridad es el que la mayoría de las personas tiene en mente cuando se está hablando de seguridad en un ordenador.

3.4.2.2 Limitación del acceso

Es preciso asegurar que un programa o un usuario no hagan más de lo que se pretende que hagan. Por ejemplo, si se ejecuta un programa descargado de la red, normalmente se desea que éste no pueda acceder al disco local para realizar ninguna operación, de forma que así no pueda dejar ningún tipo de virus o tomar ninguna información privada.

Este aspecto de seguridad está siendo tomado en mayor consideración últimamente, con el uso programas que se descargan de la red (*applets* Java, controles binarios ActiveX, etc.)

3.4.2.3 Concesión de acceso

En ocasiones es deseable permitir que dos o más personas trabajen juntas en un documento, o bien es necesario conceder acceso a un fichero a otra persona para poder delegarle trabajo.

En general se necesita una manera selectiva de conceder estos accesos, que no implique conceder más acceso del estrictamente necesario. Por ejemplo, que sea posible conceder a un colega acceso a un artículo que se desarrolla en conjunto sin que esto implique que también pueda acceder a datos personales.

3.4.2.4 Revocación de acceso

Una vez permitido el control de acceso a un objeto, es deseable tener la capacidad para eliminar ese permiso, por ejemplo cuando la persona ya no tenga autorización volver a acceder a un documento particular. Por ejemplo en el caso de un empleado que abandona la empresa o pasa a otra sección de la misma.

Aunque aquí se han explicado estos términos usando personas como ejemplos, en un sentido estricto, el control de acceso se ocupa de controlar qué es a lo que puede acceder una entidad de computación del sistema (programa), no a lo que puede acceder una persona. En un sistema informático son los programas los que acceden a las entidades, no las personas. Comúnmente se establece el concepto de identidad de usuario (una misma persona puede tener muchas identidades o cuentas diferentes) y se hace que los programas se ejecuten por orden de un usuario determinado.

3.5 Conceptos de seguridad y protección

Dentro del campo del control de acceso, o seguridad en el uso de recursos y servicios, se van a definir los conceptos de seguridad y protección, que aunque en ocasiones se utilizan de forma indistinta, sin embargo, presentan ciertos matices que introducen diferencias entre ellos.

Es posible establecer una distinción más exacta entre los términos seguridad y protección [SS94]. En el aspecto de seguridad, al igual que en algunos otros aspectos de los sistemas operativos, conviene distinguir entre la política que determina el comportamiento y el mecanismo que implementa o lleva a cabo dicho comportamiento.

El término **seguridad** suele emplearse para referirse a una política, que indica cuáles son los accesos permitidos entre los recursos. El término **protección** se refiere al mecanismo que controla el acceso a los recursos, sobre el cual se implementa la política de seguridad.

La seguridad decide, por ejemplo, qué usuarios tienen que tener acceso a qué recursos. Establece las **normas o reglas** que rigen el funcionamiento del sistema en este aspecto.

La protección se refiere a los mecanismos para controlar el acceso de los usuarios a los recursos del sistema. La protección proporciona los **medios** que permiten que se lleven a cabo las normas impuestas por la política de seguridad. Por ejemplo, un mecanismo de protección de ficheros puede consistir en un bit en cada fichero y para cada usuario. Cuando un usuario accede a un fichero, el mecanismo de protección consulta su bit asociado y si está activo permite el acceso y en caso contrario lo deniega. Una política de seguridad indicaría qué usuarios estarían autorizados para acceder a un determinado fichero (activando el bit respectivo). Como muestra, una política concreta puede determinar que sólo los usuarios pertenecientes al grupo "profesores" pueden acceder a ficheros de tipo "exámenes". La puesta en práctica de esta política (política obligatoria) conllevaría activar dentro del mecanismo de protección los bits correspondientes a los usuarios "profesores" únicamente. A partir de ese momento el mecanismo de protección asegura que los accesos permitidos llevan a cabo esa política. En las políticas discrecionales cada usuario puede libremente decidir qué accesos permite para los objetos de su propiedad. Estas últimas son las más utilizadas en sistemas de propósito general, como Unix.

Dado que las políticas pueden variar a lo largo del tiempo, el mecanismo de protección debe ser lo suficientemente flexible para acomodar un amplio abanico de políticas de seguridad. Es recomendable que el mecanismo de protección sea lo más independiente posible de la política de seguridad, permitiendo así que sea reutilizado con diferentes políticas.

No siempre, sin embargo, ambos términos están totalmente delimitados. En ocasiones existe un mecanismo sobre el que pueden imponerse diferentes políticas, pero la implantación de las políticas requieren una serie de estructuras y algoritmos que tienden a confundirse con el mecanismo básico. Muchos diseños de sistemas operativos no establecen esta división, y por tanto las funciones se mezclan.

Hablando en sentido amplio el término seguridad suele utilizarse para tratar el problema en general. Sin embargo, en muchos casos se utiliza de manera amplia el término protección para referirse a ambos aspectos ya que en ocasiones es difícil separar la política del mecanismo al estar íntimamente relacionados.

3.6 Taxonomías de seguridad

Es importante poder clasificar el nivel de seguridad ofrecido por los sistemas informáticos para tener una idea de su fiabilidad. Existen tres taxonomías [Gos91] que pueden ser usadas para este propósito.

3.6.1 Clasificación en niveles de seguridad del Departamento de Defensa de EE.UU

La primera clasificación fue introducida por el Departamento de Defensa de los Estados Unidos [DoD83]. Proporciona un mecanismo que puede ser usado para distinguir entre los sistemas "más seguros" de aquellos que son "menos seguros". El Criterio de Evaluación de Computadores, también llamado Libro Naranja, introdujo la **Base de Computación Fiable** (BCF). La Base de Computación Fiable (TCB, *Trusted Computing Base*) está compuesta por el conjunto de programas del sistema de computación que se ocupan de implementar todos los aspectos relacionados con la seguridad del sistema, garantizando la seguridad del mismo (política de seguridad y mecanismo de protección). Se denomina así porque el usuario debe confiar en que estos programas funcionan de manera adecuada, implementando la seguridad. Si esta base está comprometida (por ejemplo un intruso llega a cambiar el programa que implementa el mecanismo de protección por otro), toda la seguridad del sistema está comprometida.

El principal objetivo del Libro Naranja fue alentar a los fabricantes a proporcionar sistemas operativos más fiables. Clasifica los sistemas en cuatro divisiones etiquetadas de la A (la más segura) a la D (insegura).

- **División D: Insegura**

  Esta división agrupa a productos que no han sido evaluados o bien sólo proporcionan una protección mínima. Algunos sistemas que no intentan implementar ninguna política de seguridad, como por ejemplo el DOS, podría considerarse que incluso no alcanzan ni esta categoría.

- **División C: Protección discrecional**

  Los sistemas de esta categoría proporcionan **protección discrecional** (el usuario escoge de manera arbitraria qué proteger y cómo protegerlo). Existen dos subcategorías:

  - **Clase C1: Protección de seguridad discrecional**

    Proporciona separación de usuarios y datos, con mecanismos para proteger ficheros de usuarios de accesos no autorizados a otros usuarios.

  - **Clase C2: Protección de acceso controlado**

    Se requieren los siguientes aspectos: control de conexión al sistema (*login*), auditoría de eventos de seguridad y aislamiento de los mecanismos de protección.

- **División B: Protección obligatoria (no discrecional)**

  En esta división, la Base de Computación Fiable se encarga de hacer cumplir la seguridad. La BCF debe comprobar todos los accesos a todos los objetos y aplicar la política de seguridad. El usuario no puede cambiar libremente los permisos de los objetos. Estos permisos deben cumplir siempre la política de seguridad definida en el sistema. Se debe implementar también el concepto de monitor de referencia (módulo a través del cual se realizan todos los accesos). Existen tres subcategorías:

- **Clase B1: Protección por etiquetas de seguridad**

  Todos los objetos deben disponer de una etiqueta que los clasifica. El sistema hace cumplir la política de seguridad permitiendo el acceso de sujetos únicamente a aquellos objetos clasificados con un nivel menor que la acreditación del usuario.

- **Clase B2: Protección estructurada**

  Estos sistemas deben ser escritos desde cero, con un modelo formal de la política de seguridad, que debe estar documentado. Es necesario identificar cada canal de comunicación usado para comunicación entre procesos que pudiera ser explotado para violar la política de seguridad del sistema.

- **Clase B3: Dominios de seguridad**

  Se introducen los dominios de seguridad. Un dominio de seguridad es una lista de usuarios o grupos con privilegios de acceso a un objeto, junto con una lista de usuarios y grupos para los cuales no se debe permitir el acceso.

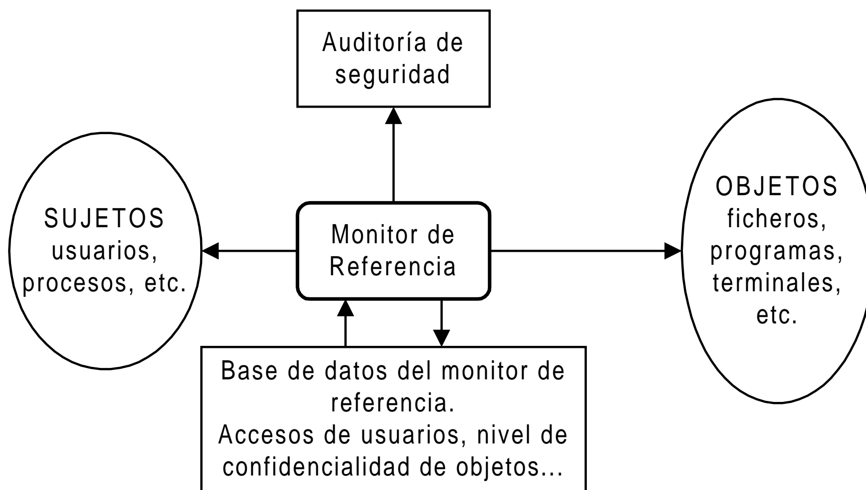


Figura 3.4: Concepto de Monitor de referencia

- **División A: Protección verificada**

  Necesita un diseño formal completo del sistema. Está orientado a información clasificada.

  - **Clase A1: Diseño verificado**

- No proporciona ninguna característica extra respecto a B3. Sin embargo, el uso de un método de diseño formal completo permite garantizar en muy alta medida que el sistema es completamente seguro.

3.6.2 Tipos de control de acceso

Una de las clasificaciones que pueden establecerse en torno a los sistemas de control de acceso distribuye éstos en dos grupos [Gos91]:

- **Control de acceso discrecional**. Cada entidad individualmente decide quién puede acceder a él y quien no. El acceso es a discreción del objeto (o del propietario del mismo). Las reglas de seguridad son especificadas a voluntad de cada objeto. Los sistemas de propósito general utilizan este tipo de control. En el sistema operativo Unix, el usuario decide cuáles son los elementos que desea proteger y contra quién los quiere proteger.

- **Control de acceso no discrecional (obligatorio)**. El acceso se permite en función de la clasificación del objeto y del tipo del sujeto que quiere acceder al mismo. Existen unas reglas de seguridad que siempre se deben cumplir. Este tipo de control suele utilizarse en sistemas más especializados, como es el caso de los sistemas militares.

3.7 Evolución del control de acceso a recursos y servicios

La seguridad proporcionada por los sistemas operativos ha ido aumentando a medida que también aumentaban los servicios que podía aportar. De modo que vamos a realizar brevemente un repaso de los recursos que un sistema puede proteger.

3.7.1 Ninguna protección

Sucede en sistemas monoprogramación donde sólo existe un programa en ejecución simultáneamente y monousuario, donde no se contempla la posible convivencia de varios usuarios en la máquina y se disponen de muy pocos mecanismos que velen por la seguridad.

Protección en el sistema operativo MS-DOS

Un ejemplo de este tipo de sistemas es el MS-DOS [Dei93]. Así, el sistema de ficheros no está protegido puesto que no existen usuarios. Cualquiera que entre en la máquina (es decir, tenga acceso físico al ordenador) es capaz de ver y modificar la estructura y el contenido de los directorios. Por otra parte, no existe tampoco protección de memoria y el programador puede acceder a cualquier posición de memoria existente. Este gran inconveniente hace que cualquier programa pueda acceder a posiciones de memoria ocupadas por el propio sistema operativo, lo que puede suponer un ataque al sistema.

3.7.2 Protección de los recursos básicos del sistema

Con la aparición de sistemas multiprogramación y multiusuario la seguridad adquiere un papel importante entre los aspectos que gestiona el sistema operativo. Los dos recursos básicos que estos sistemas deben proteger son la memoria y los ficheros.

3.7.2.1 Protección de Memoria

Cuando existen varios procesos simultáneamente cargados en memoria, el sistema debe asegurarse de que cada uno tiene asignado una zona determinada de memoria (espacio de direcciones) y que un proceso no puede acceder a direcciones de memoria situadas fuera de su propio espacio de direcciones. De lo contrario podría alterar los datos correspondientes a otros procesos corrompiendo su normal funcionamiento.

Protección hardware

El mecanismo habitual que ofrecen los sistemas para proteger la memoria es un mecanismo hardware, basado en la comprobación de cada referencia a memoria. La dirección a la que se hace referencia debe estar dentro de los límites que marca el espacio de direcciones. Según el tipo de gestión de memoria, varía ligeramente el mecanismo, pero se fundamenta en dos registros: registro base, de comienzo de la zona de memoria y registro límite, que sumado al registro base indica el límite de la zona referenciable. Basta con comprobar que la posición de memoria referida está dentro de los límites de la zona referenciable.

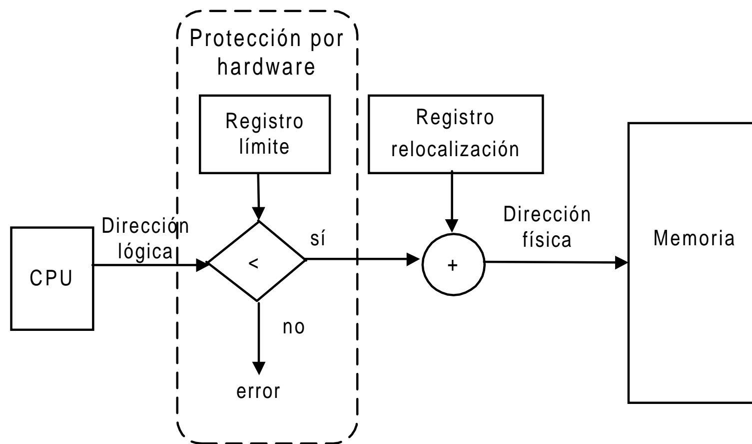


Figura 3.5: Protección por hardware

Protección software

Existen tres técnicas comúnmente utilizadas de protección software de memoria que garantizan que un programa no acceda a memoria o ejecute código para el cual no está autorizado [WBD+97]. Se trata de las siguientes: aislamiento de fallos por software, código que soporta pruebas y lenguajes seguros en tipos.

- **Aislamiento de fallos por software** [WAB+94] *(Software fault isolation).* La idea es similar a la protección hardware: las operaciones potencialmente peligrosas son comprobadas dinámicamente por seguridad antes de su ejecución. Para ello se introducen instrucciones que validan los accesos a memoria. Este mecanismo introduce entre un 5 y un 30% de sobrecarga.
- **Código con auto-demostración** [NL96] *(Proof-carrying code).* Elimina la sobrecarga producida en el aislamiento por software realizando una comprobación estática, en el momento de la carga, de que el programa respeta la política de seguridad. La principal dificultad de este método es la generación de la demostración de que el código cumple la política. Existen investigaciones en marcha para realizar la generación de demostraciones de forma automática, e incluirlo en el proceso de compilación.
- **Modelo de protección basado en el lenguaje** [Lev84]. Un ejemplo de ello puede ser Java. El acceso a memoria puede ser controlado con un lenguaje seguro, siempre que el lenguaje tenga un mecanismo para asegurar los límites de cada abstracción. Por ejemplo, en Java esto significa asegurar que las variables y métodos privados no son accesibles fuera de su clase. En este caso el sistema puede forzar el control a través de comprobaciones de seguridad antes de que se ejecute cualquier código peligroso. La comprobación de tipos de Java es estática, aunque también podría existir una comprobación dinámica, más flexible, aunque más costosa.

Niveles usuario/supervisor

Otro modo de protección básico que se ha venido utilizando habitualmente es el uso de un mecanismo hardware que establece dos modos de ejecución: el normal y el privilegiado (supervisor), de forma que instrucciones comprometidas que no debe ejecutar el usuario sólo pueden ejecutarse en modo privilegiado, exclusivo del sistema.

Un ejemplo de este tipo de instrucciones que no es recomendable que puedan ser utilizadas directamente por los programas de usuario puede serlo las de activación y desactivación de interrupciones, o las de acceso a puertos de comunicación con dispositivos como el disco duro. Normalmente es el sistema operativo el único que se ejecuta en modo

supervisor y por tanto el que puede acceder a estas instrucciones especiales cuyo uso inadecuado podría comprometer la seguridad del sistema. Cuando un programa de usuario necesita utilizar estos recursos (por ejemplo acceder a un fichero en disco duro), lo debe hacer indirectamente a través del sistema operativo, que garantiza que el acceso se hace de manera adecuada.

3.7.2.2 Protección de Ficheros

Otro recurso básico a proteger en un sistema multiusuario es el sistema de ficheros (en los sistemas que proporcionan esta abstracción, que son la mayoría). Cada usuario desea disponer de un lugar donde almacenar sus datos en forma de ficheros y que éstos puedan mantenerse a salvo de las miradas de cualquier intruso.

Protección de ficheros en el sistema operativo Unix

Un ejemplo típico es el caso del Sistema Operativo Unix [Bac86], donde el usuario puede establecer permisos sobre sus ficheros, identificando accesos de lectura, escritura o ejecución sobre cualquiera de sus ficheros, y asignando estos permisos a ninguno, algunos o todos los usuarios del sistema. De esta manera el usuario puede guardar la información con la certeza de que, si así lo desea, nadie más pueda tener acceso a ella.

El mecanismo de protección de ficheros de Unix establece tres grupos de usuarios. Los ficheros, que son las entidades a proteger, disponen de una lista de control de acceso con tres entradas, una guarda el identificador del usuario propietario del fichero, junto con sus permisos de acceso. La segunda guarda el identificador del grupo del usuario propietario con los permisos concedidos para todos los usuarios del grupo, y la tercera guarda simplemente los permisos concedidos al resto de los usuarios.

El sistema operativo Unix establece tres tipos de permisos para cada fichero: lectura, escritura y ejecución. Además de ello, el propietario del fichero y el superusuario (usuario *root* o administrador del sistema) tienen autoridad para modificar los permisos de esta lista de control de acceso.

	propietario			grupo			resto			
D	r	w	x	r	w	x	r	w	x	setUID

Figura 3.6: Máscara de protección del sistema operativo Unix

Mediante este mecanismo, el sistema restringe el acceso a los recursos de tipo fichero permitiendo un acceso selectivo de los mismos. Dado que Unix realiza las operaciones sobre recursos como dispositivos a través de ficheros representantes de dichos recursos, aprovecha el mismo mecanismo para el control de acceso a dispositivos, solucionando de este modo el problema de la seguridad en el uso de los dispositivos.

3.7.3 Protección de servicios generales del sistema y de los propios usuarios

La evolución hacia los sistemas distribuidos y las arquitecturas cliente/servidor [Tan92], han creado la necesidad de ampliar del ámbito de la protección. En este tipo de arquitecturas, la aplicación servidora ofrece unos recursos al cliente. Sin embargo, no todos los usuarios tienen por qué beneficiarse del recurso, sino sólo los que tengan permiso para ello.

Aparecen diversos modos de proteger el acceso, algunos de ellos ligados únicamente al lenguaje de programación y al enlace del código objeto. Otros sin embargo, van asociados al propio sistema operativo que controla el acceso dinámicamente en el instante de la invocación.

La protección entre recursos proporcionados por el propio usuario se convierte en un aspecto cada vez más importante en los nuevos sistemas de computación. La masiva implantación de aplicaciones y entornos orientados a objetos intensifica este problema, al facilitar que los usuarios codifiquen y proporcionen servicios. La explosión de la Internet y con ella la proliferación de códigos móviles, que pueden ser descargados de la red sobre la marcha, es en este momento un aspecto prioritario, en el que la seguridad cobra especial relevancia, y al que aún no se le han dado soluciones definitivas. La investigación en aspectos de seguridad sobre estos entornos está aún en sus inicios. Comienzan a surgir diferentes modelos de protección que intentan darle solución.

La aparición de Internet y la plataforma Java [KJS96] con el desarrollo de elementos como los *applets* (programas en lenguaje Java que viajan por la red desde cualquier servidor y se ejecutan dentro del navegador de la máquina cliente), ha abierto un nuevo agujero de seguridad que podría causar grandes problemas. Es necesario proteger los recursos locales, básicamente el sistema de ficheros, para que el código de los *applets* no pueda dañar o acceder a los datos locales.

Como puede verse, los aspectos que debe tratar la seguridad de recursos y servicios se hacen cada vez más complejos en el sistema.

Cooperación sin vulnerabilidad

En un sistema con interoperabilidad de objetos distribuidos, el objetivo de la seguridad debe ser lograr protección mientras se permite que ocurran ciertas cooperaciones. Dicho de otro modo, el problema de la seguridad consiste en cómo obtener beneficios de forma segura de la interacción con entidades no fiables.

3.7.4 Situación actual: control heterogéneo

Haciendo un resumen de la situación actual de los sistemas respecto a la seguridad se podría concluir diciendo que se utilizan diversos mecanismos *ad hoc*, específicos para el tipo de recurso a proteger. Por ejemplo, para proteger los programas en memoria entre sí se utiliza un esquema de memoria virtual con los programas en espacios de direcciones distintos. Para el control del acceso a ficheros se suele utilizar un esquema similar al de Unix. Para controlar el acceso a servidores en un sistema cliente/servidor otro totalmente diferente a los anteriores. Otro nuevo esquema aparece para los *applets* descargados de la red, etc.

Con el aumento de la interoperabilidad y la tendencia hacia los sistemas distribuidos, la situación se ha complicado considerablemente con respecto a la seguridad. Los mecanismos que se emplean para mantener el sistema seguro son variados, según el elemento a proteger y no se mantiene una homogeneidad en todo el sistema.

3.7.4.1 Problemas de la situación actual

Esta falta de uniformidad debida a la proliferación de mecanismos heterogéneos complica considerablemente la aplicación de la seguridad, haciéndola más vulnerable a fallos y difícil de utilizar.

Problemas derivados del tamaño de la Base de Computación Fiable

Por una parte, la Base de Computación Fiable (BCF) que controla el acceso a los recursos es de tamaño grande, ya que cada tipo de recurso añade su propia BCF a las que se necesitan para los otros recursos. Cuanto mayor sea la BCF, esta es más difícil de depurar, y es más probable que se produzcan errores de codificación que introduzcan agujeros de seguridad en el sistema.

Problemas de interacción entre mecanismos de protección

Por otro lado la interacción de los diferentes esquemas de protección de recursos puede hacer que se dejen "huecos" entre ellos, por interacciones imprevistas que hagan que se anulen mutuamente, por ejemplo. Incluso que se produzcan incompatibilidades que haga que no puedan utilizarse los diferentes mecanismos a la vez.

Problemas de utilización de los mecanismos por el usuario

Por último, el usuario debe aprender demasiadas estrategias de uso y conceptos diferentes para utilizar cada uno de estos distintos mecanismos. Esto puede provocar que el usuario simplemente no haga caso de estos mecanismos y no los utilice por la inconveniencia que le produce la complejidad de su uso.

3.8 Dominios de protección

Sujetos y objetos

Un ordenador puede considerarse como un conjunto de objetos y sujetos. Por objetos se entienden objetos hardware tales como CPU, segmentos de memoria, impresoras, discos, etc. y elementos software, tales como ficheros, programas, semáforos, etc. Por sujetos se entiende a las entidades activas que usan los objetos, normalmente procesos o usuarios.

La esencia de un mecanismo de protección consiste en que los sujetos deben tener permitido el acceso únicamente a aquellos recursos para los que han sido autorizados, y únicamente para unos usos determinados (sólo unos determinados tipos de operaciones del sujeto). Por ejemplo, un proceso embellecedor de listados, debería tener acceso a dos recursos: el fichero a listar y la terminal donde debe imprimirlo, y en concreto únicamente a la operación de lectura del fichero, puesto que para listarlo no necesita ni escribir ni borrar el mismo.

Sujetos y objetos en un sistema integral orientado a objetos

En el caso de sistemas integrales orientados a objetos[1], la única abstracción que existe en el sistema es el concepto de objeto. Así pues, en lo que a protección se refiere, tanto los objetos (con la definición anterior) como los sujetos son objetos del sistema. El sujeto consiste en un objeto sobre el cual se invoca una operación. La entidad activa es el objeto que inicia la invocación de la operación sobre el objeto anterior. Para evitar confusiones pueden utilizarse las denominaciones de (objeto) servidor y (objeto) cliente, respectivamente.

Dominios de protección

Para llevar a cabo la tarea anterior, suele utilizarse (aunque sólo sea conceptualmente) una abstracción denominada **dominio de protección**. Un proceso opera siempre dentro de un dominio de protección determinado, el cual especifica los recursos a los cuales puede acceder el proceso. Cada dominio establece un conjunto de objetos y los tipos de operaciones que puede realizar sobre cada objeto. Es decir, los privilegios (derechos de acceso) de un proceso están limitados a los que defina el dominio bajo el que esté funcionando el proceso.

[1] En el capítulo 5 se realiza una descripción detallada del concepto de Sistema Integral Orientado a Objetos

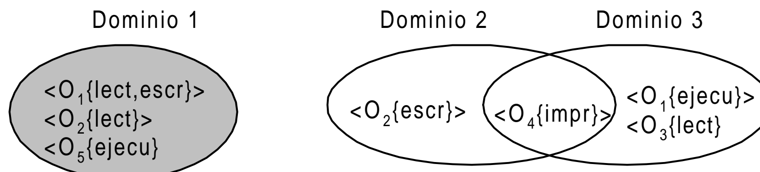


Figura 3.7: Representación de dominios de protección

3.8.1 Contextos de desarrollo de un dominio (cambios de dominio)

Los dominios pueden desarrollarse en tres contextos diferentes [Sil98]:

- **Asociados a usuarios**. El conjunto de objetos que pueden ser accedidos depende de la identidad del usuario. Cuando se produce un cambio de usuario se producirá un cambio de dominio. Este suele ser el caso de los sistemas basados en listas de control de acceso como mecanismo de protección.
- **Asociados a procesos**. En este caso, el conjunto de objetos que pueden ser accedidos depende de la identidad del proceso. Un cambio de dominio corresponde a un proceso que envía un mensaje a otro proceso, y espera por la respuesta. Mientras esté trabajando el segundo proceso, funcionará bajo su propio dominio, diferente del anterior.
- **Asociados a procedimientos**. En este caso, el conjunto de objetos que pueden ser accedidos se corresponde con las variables locales definidas dentro del procedimiento, que apuntarán a los objetos a los que se puede acceder. El cambio de dominio se produce cuando se realiza una llamada a un procedimiento. Al entrar en el procedimiento, existirá un nuevo conjunto de variables locales que definen un nuevo conjunto de sujetos a los que se pueden acceder (un nuevo dominio). Este caso (y el anterior, que puede considerarse una variante de este) se corresponde con los sistemas que utilizan capacidades como mecanismo de protección.

3.8.2 Dominios en el sistema operativo Unix

En el sistema operativo Unix el dominio está asociado con el usuario, de manera que cuando se ejecuta un proceso los permisos que éste tenga sobre los ficheros a los que desee acceder dependerán del usuario que haya ejecutado el proceso. Es decir, el proceso puede acceder a todos los recursos a los que pueda acceder el usuario bajo el cual se ejecuta el proceso (en Unix los permisos de acceso a los ficheros se definen en función de los usuarios).

El sistema operativo proporciona un mecanismo para realizar un cambio temporal de dominio. Se trata del uso del bit *setuid*, que cambia el usuario efectivo de un proceso (el usuario que el sistema utiliza para la comprobación de permisos) [Bac86]. Si un usuario ejecuta un programa cuyo bit *setuid* está activo, al ejecutarse el sistema toma como usuario efectivo el propietario del programa y por tanto el proceso se ejecutará con TODOS los permisos de ese propietario (el dominio pasa a ser el correspondiente al propietario del fichero que se ejecuta, no el del usuario actual). Así si el programa pertenece al sistema el programa podrá acceder a ficheros del sistema aunque el que lo ejecute sea un usuario normal.

Esto es útil en programas como el ***passwd*** que ejecuta un usuario cualquiera y le permite cambiar su contraseña de acceso. Para ello este programa tiene que acceder al fichero de contraseñas, que es propiedad del superusuario (obviamente, ya que si no un usuario cualquiera podría cambiar las contraseñas del resto). Haciendo que el programa ***passwd*** sea propiedad del superusuario y activando el bit *setuid* de este programa, se permite que un

usuario cualquiera pueda utilizar el programa ***passwd*** y no pueda cambiar arbitrariamente el fichero de contraseñas. Al ejecutar el programa ***passwd***, el usuario efectivo es el superusuario, con lo que el sistema permite el acceso al fichero de claves propiedad del mismo. Un usuario cualquiera puede actualizar el fichero de contraseñas sólo de forma controlada a través de este programa.

Este mecanismo lo que permite es una **amplificación de derechos** temporal y controlada de un usuario normal. Sin embargo, esta amplificación de derechos es de grano tan grueso que puede provocar grandes problemas de seguridad. El problema es que el proceso que tiene el bit *setuid* toma todos los permisos del usuario efectivo, aunque normalmente no se necesitará más que una pequeña parte de los mismos. Esta granularidad gruesa hace que por ejemplo el programa ***passwd***, que en realidad sólo necesita acceder al fichero de contraseñas, potencialmente puede hacer cualquier cosa, como por ejemplo borrar todo el sistema de ficheros. Un fallo en la codificación de este programa puede ser aprovechado por un intruso para introducir caballos de Troya, ganar permisos de superusuario, dañar el sistema, etc. Este problema en la granularidad excesiva en la amplificación de derechos es uno de los focos mayores de agujeros de seguridad en el sistema Unix[1] y en general de los sistemas basados en listas de control de acceso.

3.9 Principios de diseño de mecanismos de seguridad

Saltzer y Schroeder, [SS75] identifican una serie de principios en el diseño de las medidas de seguridad para las diversas amenazas a los sistemas informáticos. Entre estos principios se incluyen los siguientes; mínimo privilegio, ahorro de mecanismos, aceptación, mediación total y diseño abierto.

3.9.1 Mínimo privilegio

Todos los programas y usuarios del sistema deben operar utilizando el menor conjunto de privilegios necesarios para completar la labor. Los derechos de acceso deben adquirirse sólo por permiso explícito; por omisión el acceso no debe estar permitido.

Obviamente, si un programa tiene más permisos de los estrictamente necesarios para realizar su tarea, existe un posible agujero de seguridad (como en el ejemplo anterior del bit *setuid* del Unix). El programa podría utilizar estos permisos “extra” para realizar tareas adicionales (posiblemente peligrosas) que en principio no debería realizar. Para evitar esto, cuando un programa necesite un cierto permiso para realizar una tarea determinada, debe concedérsele explícitamente y sólo ese permiso.

3.9.1.1 Control de “Caballos de Troya”

Un ejemplo es el de un programa que simplemente lista un fichero. Para realizar su tarea únicamente necesita permiso de lectura sobre el fichero que hay que listar. Si por omisión se le concede por ejemplo todos los permisos sobre el fichero, este programa (que sólo debería poder listar) podría borrar el fichero. Esto es especialmente importante para evitar caballos de Troya, programas que el usuario no ha desarrollado y por tanto no sabe cómo están implementados y que hacen cosas diferentes a las que aparentemente debería hacer (normalmente dañinas). Más importante es ahora, si cabe, con la proliferación de programas que se descargan de la red (como en el caso anterior, sería fácil camuflar un caballo de Troya

[1] No sólo de Unix, si no de la mayoría de sistemas operativos basados en estos mismos conceptos, y de amplia difusión, como Windows NT, Windows 95, Novell, VMS, etc. (si bien unos en mayor medida que otros).

en el programa de listados). El principio de mínimo privilegio permite que estos programas sólo tengan privilegios para realizar lo que anuncian como su funcionalidad.

3.9.2 Ahorro de mecanismos

Los mecanismos de seguridad deben ser tan pequeños y simples como sea posible, ayudando a su verificación. Esta exigencia suele suponer que deben ser una parte integral del diseño del sistema, más que mecanismos añadidos a diseños existentes.

En efecto, cuanto más sencillo y pequeño sea el mecanismo de seguridad (es decir, la Base de Computación Fiable), más pequeña será la implementación del mismo y será más fácil de entender por el usuario. Una implementación pequeña es más fácil de depurar, a la vez que es más sencillo verificar que funciona correctamente. Así se evitan agujeros de seguridad derivados de una implementación defectuosa.

3.9.2.1 Diseño del mecanismo de seguridad como parte integral del sistema

Cuando los mecanismos de seguridad se añaden a sistemas existentes (“parches”), la difícil integración con lo existente hace que sean más complicados conceptualmente y que aumente el tamaño de su implementación, con los problemas mencionados anteriormente. Esto hace que la mejor manera de respetar este principio sea diseñar el mecanismo de seguridad al mismo tiempo que el resto del sistema en su conjunto, formando parte integral del mismo, y no siendo un “parche” añadido posteriormente.

3.9.3 Aceptación

Los mecanismos de seguridad no deben interferir excesivamente en el trabajo de los usuarios, mientras cumplen al mismo tiempo las necesidades de aquellos que autoricen el acceso. Si los mecanismos no son fáciles de usar, probablemente no van a ser usados o lo serán de forma incorrecta.

3.9.3.1 Mecanismo de seguridad conceptualmente sencillo

Un mecanismo de seguridad muy bueno, pero que no sea utilizado por el usuario, o lo sea incorrectamente no cumple su función. Por una parte, el mecanismo debe ser conceptualmente fácil de entender (ahorro de mecanismo en el nivel conceptual) para que el usuario pueda aplicarlo correctamente. Si no es así, se corre el riesgo de que el usuario lo aplique incorrectamente, y lo que es peor, posiblemente en el convencimiento de que está haciéndolo bien y se encuentra protegido.

3.9.3.2 Mecanismo de seguridad fácil de usar

Por otro lado el mecanismo debe ser fácil de usar y no molestar al usuario en su trabajo hasta el punto de que este no haga caso al mecanismo. Por ejemplo, si antes de la ejecución de un programa, un mecanismo de seguridad requiere que se consulte al administrador del sistema para que verifique el funcionamiento del programa, esto sería tan incómodo para el usuario que acabaría saltándose este paso si fuera posible, ejecutando los programas sin ninguna protección[1].

[1] Un ejemplo del peligro de los mecanismos incómodos es el caso común de limitar los programas que se pueden ejecutar a una serie de programas autorizados por la dirección. Al final los usuarios invariablemente acaban ejecutando otros programas (juegos, etc.) si tienen ocasión. Otro ejemplo es el de los rastreadores de virus que había que pasar manualmente antes de ejecutar cualquier nuevo programa. Tarde o temprano se acaba por olvidar su activación por la incomodidad que representa.

3.9.4 Mediación total

Cada acceso debe ser cotejado con la información de control de acceso, incluyendo aquellos accesos que suceden fuera de la operación normal, como la recuperación y el mantenimiento del sistema.

3.9.4.1 Imposibilidad de esquivar el mecanismo de seguridad

Este principio indica que absolutamente todo el funcionamiento del sistema debe ser controlado por el mecanismo de seguridad. Ninguna parte del sistema, en ningún momento, puede funcionar al margen del mecanismo de seguridad. De esta manera se asegura la integridad del sistema en todo momento, ya que ningún acceso deja de ser controlado por el mecanismo.

3.9.5 Diseño abierto

La seguridad del sistema no debe depender de guardar en secreto el diseño de sus mecanismos. De esta forma, los mecanismos podrán ser revisados por muchos expertos y los usuarios podrán, por tanto, depositar una alta confianza en ellos.

3.9.5.1 Peligro del probable descubrimiento de un diseño secreto

Hacer depender la seguridad de un sistema del secreto de su diseño es una mala elección. Tarde o temprano se acaban por descubrir estos secretos de diseño. El diseño de un sistema es una tarea compleja que involucra a muchas personas y genera mucha documentación. Es muy difícil tener controlada toda esta información de diseño. Además, mediante técnicas de ingeniería inversa de la implementación, gran parte del funcionamiento del sistema puede ser averiguado. En el momento en que el secreto de diseño deja de ser tal, el sistema pasa a ser totalmente inseguro, y lo que es peor, probablemente sin que lo sepan sus usuarios, que quedan totalmente desprotegidos.

Es mucho más inteligente utilizar un mecanismo que se base en un diseño público, que por otro lado podrá ser comprobado por muchas más personas, incluso ajenas al desarrollo del sistema. Así es más sencillo detectar posibles defectos en el diseño del sistema, a la vez que existe una confianza mayor en que el sistema funciona correctamente.

3.10 Problemas que debe resolver un sistema de seguridad

Actualmente, los límites de lo posible y lo imposible, no han sido bien delimitados en el campo de la seguridad en computación. Se necesita una mayor organización y formalización de los problemas relacionados con la seguridad de los sistemas. Esta organización constituye un aspecto fundamental a la hora de delimitar la bondad de un sistema de seguridad.

Cooperación sin vulnerabilidad

El objetivo de la seguridad debe ser lograr seguridad mientras se permite que ocurran ciertas cooperaciones. Dicho de otro modo, el problema de la seguridad consiste en cómo obtener beneficios de forma segura de la interacción con entidades no fiables [Mil99].

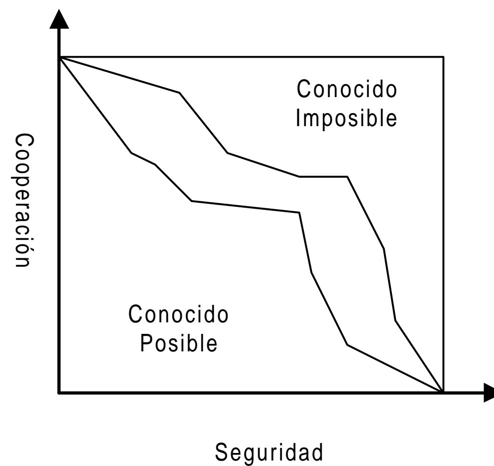


Figura 3.8: Gráfico de "cooperación sin vulnerabilidad"

La seguridad completa se logra cuando no es necesaria la cooperación y por tanto todas las entidades pueden aislarse de las demás. A mayores niveles de cooperación, más dificultades tiene el sistema para mantener los niveles de seguridad. En una cooperación con libertad total es imposible la seguridad completa.

Existen múltiples tipos de seguridad y muchos patrones de cooperación. Algunas de estas combinaciones son deseables, y son conocidas como posibles en computación. Otras son conocidas como imposibles. Entre ambas zonas existe una zona amplia que se corresponde con lo desconocido. Esta zona no ha sido demasiado tratada en la bibliografía existente.

Una vez establecidos y formalizados los límites de vulnerabilidad en entornos cooperantes, pueden aplicarse a los distintos modelos y arquitecturas de seguridad existentes. Un modelo de seguridad podrá expresar ciertas relaciones de forma segura, ciertas prohibiciones, y fallará en la expresión de otras. Este análisis permitirá realizar críticas respecto a la expresividad del modelo. Un modelo puede fallar al proporcionar una expresión de seguridad que sea tanto posible como deseable, y puede posibilitar la expresión de seguridad que es imposible. El primero será débil, el segundo peligroso. La expresión de prohibiciones que no se pueden prevenir.

3.10.1 El problema de la fuga de información

La fuga de información (o más correctamente, de datos) ocurre cuando alguien o algo es capaz de obtener información que no debería poder obtener. Por ejemplo, el contenido de un fichero que se supone que no puede ser leído.

Fuga de autoridad

La autoridad es el derecho que se tiene a realizar cierta operación sobre un objeto determinado. La fuga de autoridad ocurre cuando alguien o algo llega a ser capaz de realizar ciertas acciones que no debería tener posibilidad de hacer. Por ejemplo, que un usuario normal consiga el permiso para modificar el fichero de contraseñas del sistema.

Causas de la fuga de información o autoridad

Cuando se produce una cooperación entre distintos elementos, es necesario controlar las fugas. Existen cuatro factores que marcan cuatro posibles problemas de fuga, que serán objeto de estudio. Antes de su descripción se analizará cuál es la situación.

Supóngase que un cliente C, desea realizar operaciones en un programa servidor S, para lo cual es necesario que C le envíe a S un fichero de datos confidenciales FD (en general un

objeto). Es posible que exista un tercer elemento (espía E) que desee obtener de alguna manera los datos confidenciales del cliente.

En términos de autoridad, el cliente le envía al servidor la autoridad para acceder al objeto de datos, y se trata de que el espía no consiga a su vez autoridad para acceder al objeto de datos.

En esta situación aparecen los siguientes casos:

- Que el Servidor S y el espía estén en comunicación o no. Es decir, que exista un canal de comunicación normal del sistema de computación (no encubierto) entre ellos. Por ejemplo, en un sistema orientado a objetos, que ambos objetos puedan llamarse métodos entre sí.
- Que el Servidor desee dar o no la información al espía E. Es decir, que el servidor y el espía estén compinchados para que el espía acceda a la información.

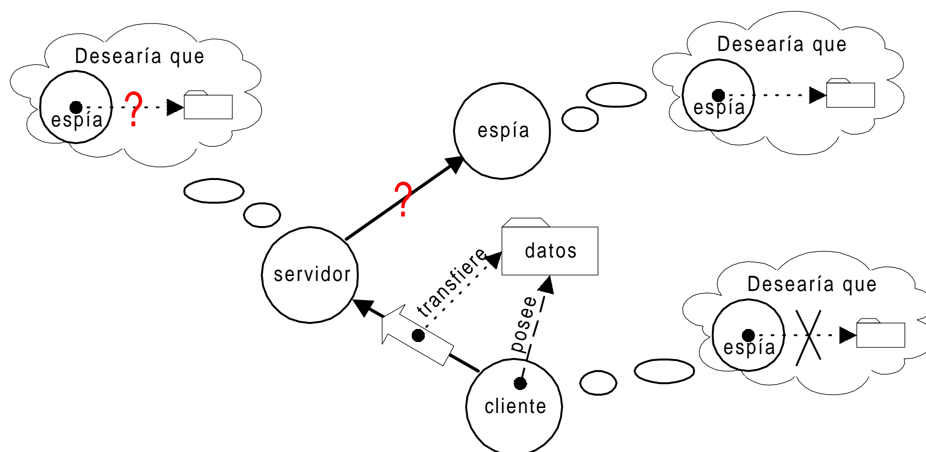


Figura 3.9: Posibilidades de fuga de información o autoridad

En función de estas cuatro posibilidades aparecen los siguientes problemas de seguridad [Eri99, Mil99].

	Servidor y espía no compinchados	Servidor y espía compinchados
Servidor y espía incomunicados	DEFENSA PERIMETRAL	CONFINAMIENTO
Servidor y espía comunicados	REPRESENTANTE CONFUSO	CONSPIRADORES EN COMUNICACIÓN

Tabla 3.1: Clasificación de problemas de fuga de información o autoridad

Esta es una clasificación que reúne diferentes problemas de seguridad que se trataban tradicionalmente por separado bajo el marco común de referencia que se acaba de presentar.

3.10.2 Defensa perimetral

La defensa del perímetro consiste en asegurar que si entre el servidor y el espía no existe ninguna línea de comunicación y el servidor no tiene intención de pasarle información al espía, éste no podrá acceder a los datos confidenciales.

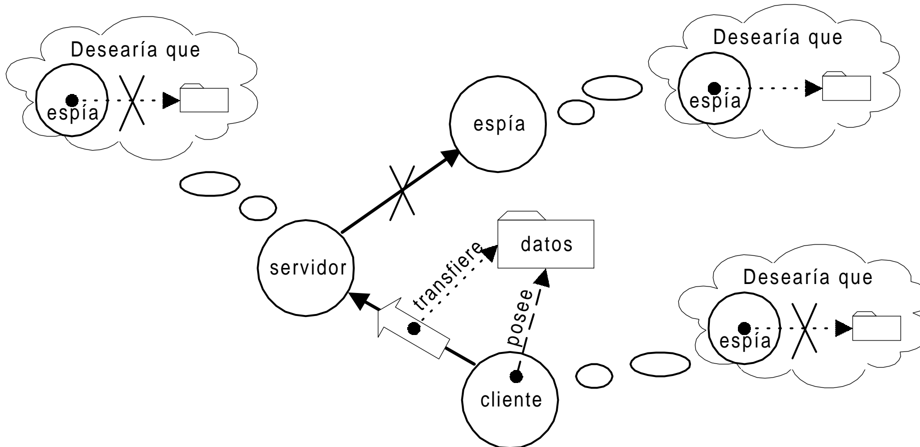


Figura 3.10: Defensa perimetral

La seguridad perimetral puede considerarse como la parte de prevención de acceso de un mecanismo de control de acceso. Por ejemplo, se trata de que un usuario "ladrón" (espía), no pueda acceder a datos bancarios que un usuario normal (cliente) confía a su "banquero" (servidor). Otro ejemplo puede ser el ataque de una persona ajena a un sistema (*hacker*) para ganar acceso a un sistema sin tener acceso inicial al mismo.

Algunas técnicas que se pueden utilizar para lograr la defensa del perímetro siguen a continuación:

- Los sistemas operativos utilizan la protección hardware de memoria, para asegurar que procesos distintos no puedan acceder a otros espacios de direcciones, y por tanto a otros datos, que no sean los propios.
- Las redes utilizan la criptografía para evitar el acceso a datos ajenos.
- Los lenguajes utilizan los verificadores (como Java), comprobación estricta de tipos y lenguajes seguros (garantizan que sólo se pueden manipular los datos a través de las operaciones definidas al efecto). Así se evita que el código pueda realizar acciones que no debería poder hacer.
- El hardware utiliza etiquetas hardware para proteger posiciones de memoria, registros de límites, etc.

3.10.3 El representante confuso

Este caso consiste en evitar que el espía se haga con el fichero de datos confidenciales del cliente a pesar de existir la posibilidad de comunicación explícita entre el servidor y el espía, y teniendo en cuenta que el servidor no desea pasar al espía la información.

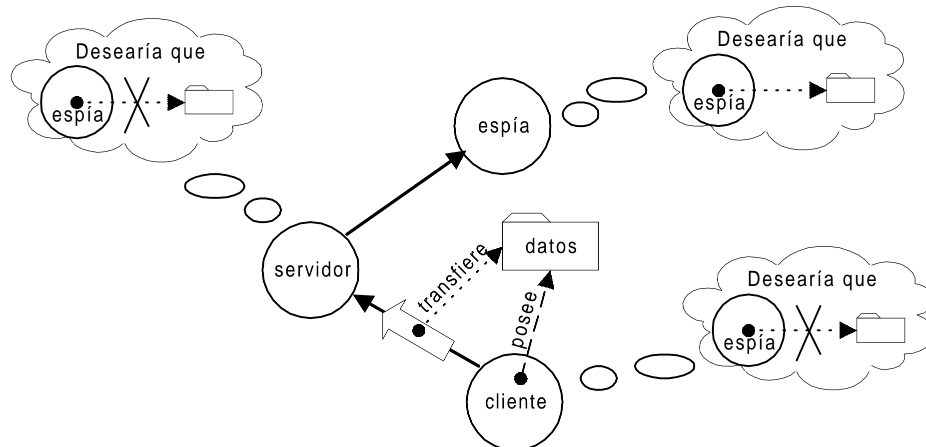


Figura 3.11: El representante confuso

Para explicar este problema se muestra un ejemplo que puede ocurrir en un sistema operativo como Unix. El problema [Har88] puede verse en el caso de un compilador que genera un fichero de estadísticas sobre utilización cada vez que es utilizado por un usuario, cuyo nombre le pasa el usuario como parámetro. Además de éste, el compilador guarda datos en un fichero denominado “factura” con información de uso del compilador, para que el administrador del compilador pase la factura al usuario por su uso.

El problema surge cuando el usuario (actuando como espía) envía como nombre de fichero de estadísticas el fichero “factura”. En ese caso, el compilador (en el papel de servidor), guarda en éste los datos estadísticos y se carga el fichero de factura real.

Este agujero de seguridad se produce porque el compilador posee simultáneamente dos autoridades, la del usuario y la del administrador del compilador. Una autoridad es la del propietario del compilador, a través del cual accede al fichero “factura”, y la otra la del usuario que lo ejecuta, a través del cual accede al fichero con el código fuente a compilar y al fichero de estadísticas. Por ello el compilador es un representante confuso. Las autorizaciones se separaron de los objetos sobre las que habían sido concedidas, y el usuario fue capaz de utilizar la autorización que se había concedido al compilador para grabar en el fichero de “factura”, y no al propio usuario.

El compilador no tiene intención de que el usuario obtenga datos del administrador. Sin embargo, el usuario (actuando como proceso malvado), consigue destruir información del administrador, basándose en el hecho de que existe un modo de comunicación del compilador con él y a la vez con el administrador.

3.10.4 Confinamiento

El segundo caso se produce cuando el servidor pretende pasar la información confidencial al espía (están compinchados), a pesar de no contar con un canal de comunicación entre ambos. Un sistema que no permita que se produzca esta situación evitará el problema del confinamiento.

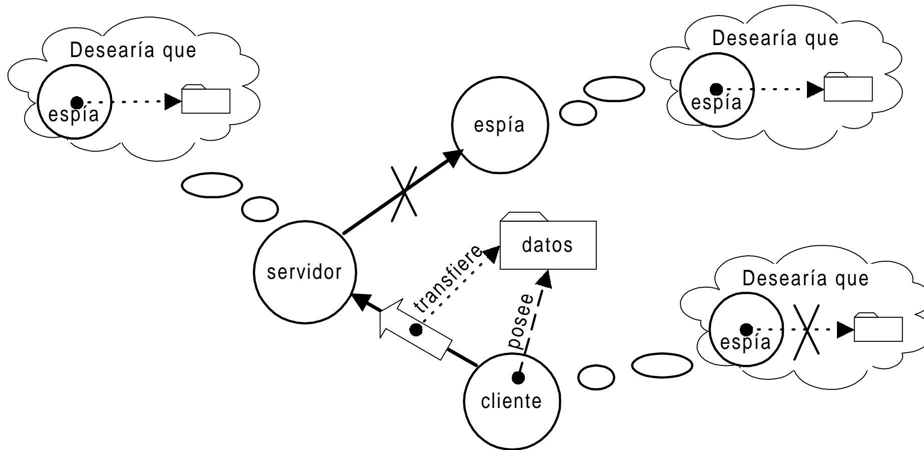


Figura 3.12: Problema del confinamiento

Canales encubiertos

A pesar de no existir un medio de comunicación directo, el servidor compinchado podría utilizar medios indirectos (**canales encubiertos** [Lam73]), para transmitir información al espía. Estos medios aprovechan propiedades incidentales de un sistema de computación que pueden ser observables por ambas partes aunque no estén directamente comunicadas.

Por ejemplo, un programa podría producir en un intervalo de tiempo determinado una tasa elevada de fallos de página, indicando con esto un bit 1 a otro programa que pueda observar la tasa de fallos. De esta manera podría transmitir información al segundo programa.

Como se ve, los canales encubiertos permiten transmitir (en principio), únicamente información. Si la información que transmiten es, por ejemplo, una contraseña, entonces podrían transmitir autoridad.

Asegurar el confinamiento evitando la posibilidad de canales encubiertos es muy costoso de implementar.

3.10.5 Conspiradores en comunicación

En el caso del representante confuso el servidor no tenía intención de pasarle información al espía, aunque exista un canal de comunicación con éste.

Existe un caso aún más extremo, en el cual el servidor se comunica con el espía y además tiene intención de pasarle información confidencial. En algunos casos, cuando se habla de confinamiento, también se hace referencia a este aspecto.

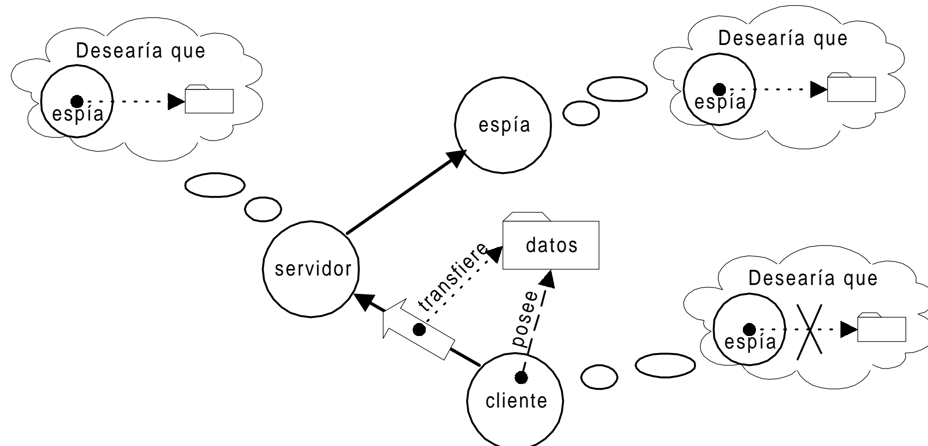


Figura 3.13: Conspiradores en comunicación

Este caso no tiene solución práctica, puesto que si el servidor tiene un canal de comunicación con el espía y además tiene la intención de ayudar al espía, no existe fuerza en el universo (de la computación) que lo impida.

Otra cuestión es evitar que el servidor pase la información al espía impidiendo la utilización del canal de comunicación entre ellos (canal normal del sistema) de alguna manera. En este caso estaríamos en el caso anterior del confinamiento, en el que no hay comunicación entre los conspiradores.

3.10.6 Otros puntos de vista de la fuga de información y de los problemas que debe resolver un sistema de seguridad

El problema de las fugas ha sido convenientemente formalizado en el apartado anterior. Hasta ahora, este problema se trataba de forma parcial, dedicando mayor cuidado a unos u otros aspectos en función de las necesidades del sistema, y con nombres que incluyen varios aspectos de los desglosados anteriormente. Normalmente se refiere exclusivamente a la parte de fuga de información y no a la de fuga de autoridad.

Por completitud y perspectiva histórica se destacan los problemas del confinamiento, revocación y delegación, considerados problemas "clásicos" de referencia dentro de la literatura. En este apartado se añaden algunos más que pueden también ser considerados.

3.10.6.1 Propagación

Consiste en evitar propagar la autoridad de manera incontrolada. Sólo aquella autoridad que deba transmitirse podrá ser transmitida. El sistema de seguridad debe asegurarse de que esto es así para poder afirmar que ejerce un control sobre la propagación.

El problema de la propagación es un caso particular del problema del confinamiento y de los conspiradores en comunicación, donde la información que se distribuye es información necesaria para el acceso a recursos.

3.10.6.2 Sospecha mutua

En muchas ocasiones los sistemas preocupados por la seguridad definen el concepto de "sospecha mutua", para referirse a la necesidad de proteger los elementos que se comunican en ambos sentidos. En un sistema cliente servidor, tanto el servidor debe protegerse de clientes malvados, como al revés. El cliente debe protegerse del servidor, el cual no necesariamente tiene que ser fiable. Esta idea es una consecuencia de la posibilidad de fuga de información que puede existir en un sistema, y de los problemas que se derivan de ella.

En ocasiones se habla de solucionar el problema de la sospecha mutua para integrar todas las posibilidades.

3.10.6.3 Delegación

Debe ser posible, temporalmente, extender los permisos del usuario para una operación específica. Es decir, delegar los permisos necesarios para realizar esa operación en el usuario durante el tiempo necesario para desempeñarla.

Este problema surge cuando el dominio de protección está asociado al usuario, es decir, cuando un programa que ejecuta un usuario tiene acceso a aquellos objetos a los que tenga acceso el usuario en cuestión. En ocasiones no es deseable que el sistema se comporte de esta manera, sino que lo deseable sería que el dominio se asociara al proceso, y por tanto se dispusiera de los permisos asociados al proceso.

El problema del Juego de Kowalski

Un ejemplo tradicional de este problema es el del juego de Kowalski [Kow90]. Se trata de un juego que mantiene una tabla de puntuaciones. Si un usuario ejecuta el juego y gana, el juego deberá acceder a la tabla de puntuaciones para añadir el nombre del usuario en la clasificación. Sin embargo, el juego no puede acceder a la tabla, ya que el usuario bajo el que se ejecuta el juego no debe tener permisos para acceder a esta tabla (pues podría modificar arbitrariamente la clasificación). Temporalmente es necesario extender los permisos del usuario y hacer que éstos sean los permisos que permiten el acceso a la tabla (del dominio necesario para realizar el juego), para que el juego funcionando bajo este usuario pueda acceder a la tabla.

Este problema, como se describió anteriormente, aparece en el sistema Unix para el cambio de contraseña de un usuario, y se intenta resolver en este sistema mediante el uso del bit *setuid* asociado a cada fichero, aunque la granularidad de la delegación es demasiado gruesa.

Este problema podría englobarse dentro del caso del representante confuso, en el que el usuario cede (delega) permiso al compilador para que acceda a las estadísticas y el administrador le cede permiso al compilador para que genere los datos de facturación.

3.10.6.4 Revocación

El problema de la revocación consiste en poder eliminar los permisos previamente concedidos a un sujeto. Un permiso concedido en un momento dado, puede necesitar ser eliminado en un futuro.

Por ejemplo, se puede conceder permiso de acceso a informes confidenciales técnicos a un determinado usuario. Sin embargo, posteriormente, el usuario pasa a otra sección de la empresa y ya no debe poder acceder esos informes confidenciales. Es necesario poder revocar el permiso de acceso que tenía anteriormente.

Este problema se trató anteriormente como uno más de los aspectos que incluye el control de acceso.

3.10.6.5 Confinamiento

Este problema fue definido inicialmente por Lampson [Lam73], aunque sólo para el confinamiento de la información. Consiste en mantener aislados los datos utilizados en una aplicación de forma que ésta no pueda utilizarlos en su beneficio para otras misiones fuera de las que les ha sido asignada. Es decir, se trata de asegurar que los datos que se entregan a una aplicación sólo se utilizan para los fines exactos que manifiesta la especificación de la aplicación, y quedan confinados a los límites de la misma.

Supongamos el siguiente ejemplo. Un programador P ha escrito dos programas que se ejecutan en la misma máquina. El programa 1 es un programa de análisis financiero, el cual es utilizado por un usuario U. P sin embargo puede intentar que los datos confidenciales recibidos de U para utilizar como entrada del análisis financiero sean enviados también a otro programa 2 y así poder utilizar estos datos a su antojo. Si no existe un medio mediante el cual confinar los datos confiados al programa de análisis financiero, un usuario no puede estar seguro de la confidencialidad de sus datos.

En este problema se engloban los problemas anteriores de confinamiento estricto (no hay un medio de comunicación directo entre el programa 1 y el 2) y el de conspiradores en comunicación (sí existe comunicación entre ellos).

3.10.6.6 Caballos de Troya

Se trata de evitar que un programa que dice que desarrolla una función determinada pueda en realidad realizar otras diferentes, normalmente dañinas para el sistema. Es decir, programas que, como el caballo de Troya, se disfrazan anunciando unas funciones (normalmente beneficiosas o cosméticas) para luego en realidad desempeñar otras muy distintas (normalmente dañinas, como instalar un virus, borrar el disco duro, capturar datos confidenciales, etc.).

Este problema ya se trató en los principios de diseño de un mecanismo de seguridad. Es un problema relacionado con la limitación de acceso de un sistema de control de acceso. La utilización del principio de mínimo privilegio previene este problema, pues permite asignar a cada programa exactamente los permisos necesarios para desempeñar la tarea que anuncia. Por tanto, no tendría permisos para hacer aquellas tareas dañinas que no anuncia.

Los sistemas que asocian el dominio de protección al usuario sufren mucho este problema, pues al ejecutar un programa (el caballo de Troya en este caso), el programa tiene todos los permisos del usuario, los necesite para realizar sus tareas “oficiales” o no y podría aprovechar estos permisos adicionales para hacer sus acciones dañinas.

4 Modelos de Seguridad

Han sido desarrollados varios modelos genéricos de protección de recursos para sistemas operativos, cuyo objetivo es controlar el acceso de los usuarios o procesos (**sujetos**) a los recursos del sistema operativo, que deben ser protegidos (**objetos**).

En un sistema operativo tradicional los objetos a proteger son recursos (ficheros, etc.) y los sujetos que acceden son procesos o usuarios. En la perspectiva de un sistema operativo orientado a objetos, ambos elementos son objetos (del propio SO o creados por el usuario). El acceso al objeto consistirá en el envío de un mensaje al mismo (invocación de una operación).

En los siguientes apartados se describen brevemente algunos de los modelos de protección más utilizados.

4.1 Modelo de la matriz de acceso

Este modelo fue propuesto por Lampson en 1971 [Lam71] como una descripción generalizada de mecanismos de protección en sistemas operativos, y mejorado por Graham y Denning en 1972 [GD72] y Harrison, Ruzzo y Ullman en 1976 [HRU76]. Se trata del modelo más utilizado, del que existen numerosas variaciones, especialmente en su implementación.

El modelo considera un conjunto de recursos, denominados objetos, cuyo acceso debe ser controlado y un conjunto de sujetos que acceden a dichos objetos. Existe también un conjunto de permisos de acceso que especifican los derechos que los sujetos pueden tener sobre los objetos (normalmente lectura, escritura, etc., aunque pueden ser diferentes, en general, dependiendo de las operaciones que puedan realizarse con el objeto).

Se trata de especificar para cada pareja (sujeto, objeto), los permisos de acceso que el sujeto tiene sobre el objeto. Esto se representa mediante una matriz de acceso *M* que enfrenta todos los sujetos con todos los objetos. Se representan los permisos para cada pareja (sujeto, objeto), indicando los permisos de acceso concretos que tiene el sujeto *i* sobre el objeto *j*.

La implantación más simple y natural de este modelo de protección puede ser en forma de matriz, donde cada entrada *M[i,j]* posee los permisos para cada pareja (sujeto *i*, objeto *j*). Esta matriz es una matriz dispersa, donde la mayoría de las celdas están vacías, ya que en general cada sujeto sólo tiene acceso a un número reducido de todos los objetos existentes en el sistema. En esta matriz las filas representan los sujetos, las columnas los objetos, cada entrada en la matriz consiste en un conjunto de permisos de acceso, y la entrada *M[i,j]* define el conjunto de operaciones que un sujeto S_i puede invocar sobre el objeto O_j

La siguiente figura representa una matriz de acceso:

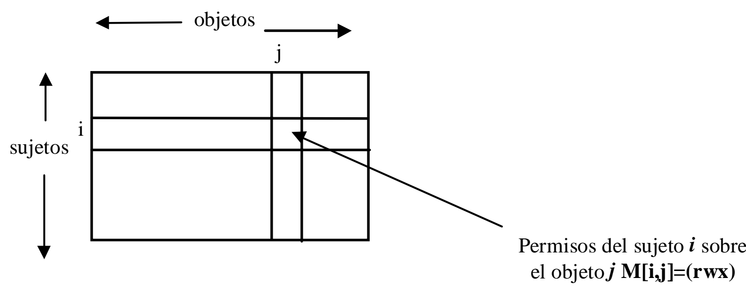


Figura 4.1: Matriz de acceso

En un sistema operativo orientado a objetos (SOOO) los permisos almacenados en *M[i,j]* serían las operaciones (métodos) que puede invocar el objeto *i* sobre el objeto *j*. El resto de las operaciones que tenga definidas el objeto *j* y no aparezcan en *M[i,j]* no pueden ser usadas por el objeto *i*. Por otra parte, todo lo que existe en el sistema son objetos, por tanto, sería una matriz cuadrada de objetos, que formaría un espacio disperso, dado que cada objeto sólo tiene acceso a operaciones de un pequeño número de objetos de todo el sistema.

4.2 Modelo de seguridad Take-Grant

Se trata de un modo de representación de la matriz de acceso de Lampson. En él se representa esta matriz mediante un grafo dirigido. Los objetos son nodos y un arco entre nodos indica que el primer objeto puede acceder al segundo. El arco se etiqueta con los permisos de acceso.

Si un objeto *X* tiene permisos sobre un objeto *Y*, existirá un nodo *X* del que sale una flecha al nodo *Y*, etiquetada con los permisos que *X* tiene sobre *Y*.

Se usan dos permisos especiales para modificar el grafo: ***take*** (tomar) y ***grant*** (conceder), que funcionan de la manera descrita a continuación. Si *X* tiene permiso *take* sobre *Y*, puede acceder a cualquier objeto sobre el que tenga acceso *Y* (puede tomar los permisos de *Y*). Si *X* tiene permiso *grant* sobre *Y*, *X* puede concederle a *Y* permisos de acceso a sus objetos (puede conceder sus permisos a Y).

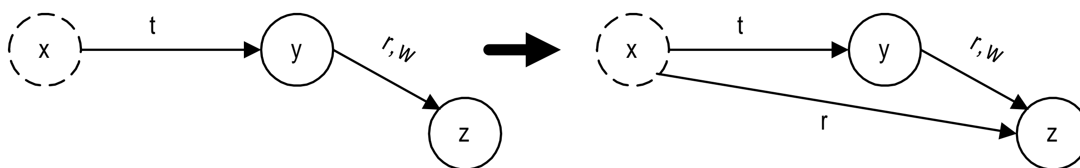


Figura 4.2: Operaciones del modelo Take-Grant

4.3 Modelos de seguridad de flujo de información

Una desventaja del modelo de la matriz de acceso es que la semántica de la información no es considerada. Es necesario, por tanto, encontrar unas restricciones adecuadas para que el sistema sea seguro.

Es posible extender los sistemas de protección para que el estado de protección sea determinado por el flujo de información segura. Este planteamiento está derivado de los modelos de seguridad militar, y se conoce comúnmente como el modelo de seguridad de retículo. Bell y LaPadula [BL73] realizaron una implementación del modelo para las fuerzas

aéreas de USA, y Denning [Den76] formalizó posteriormente el modelo, ampliando su descripción Landwehr [Lan81] y Maekawa [MOO87].

El modelo en retículo es una extensión del modelo de la matriz de acceso, incluyendo acreditaciones, clasificaciones y reglas respecto a la clasificación. En este tipo de modelos se hace una clasificación de los objetos en tipos, y de los sujetos en categorías (o acreditaciones), y en función de ello se establecen controles de acceso, representados mediante reglas que definen una ordenación con unas propiedades especiales, para mantener la seguridad. En función de estas reglas se obtendrán diferentes modelos.

Los sujetos tienen acreditaciones como capitán general, coronel, etc. y los objetos se clasifican en confidencial, alto secreto, etc. Las reglas indican bajo qué condiciones un sujeto con una acreditación dada puede acceder a los objetos de cada clasificación, por ejemplo a qué objetos puede acceder un coronel, etc.

Modelo de seguridad en retículo de Denning

Este modelo está basado en el concepto de seguridad en retículo. El modelo requiere que toda la transferencia de información esté sujeta a la relación de flujo entre las clases de seguridad. El elemento básico es un flujo de información de un objeto a otro más que un acceso individual a los objetos.

El modelo de seguridad formalizado por Denning está basado en la siguiente definición de modelo de flujo de información:

$FM=\langle N,P,SC,\oplus,\rightarrow\rangle$, *donde*

- *N* es el conjunto de objetos de almacenamiento lógico (ficheros, segmentos, variables, etc.)
- *P* es un conjunto de procesos, que representan sujetos responsables del flujo de información.
- *SC* es un conjunto de clases de seguridad, que responden a clases de información disjunta. Cada objeto pertenece a una clase de seguridad. Los usuarios pueden tener limitadas las clases de seguridad, referidas como “acreditaciones de seguridad”, y, por tanto, cada proceso.
- $\oplus$ es un operador de combinación de clases, de tipo binario, asociativo y conmutativo, que especifica, para cada par de clases operando, la clase a la que pertenece el resultado de la operación con valores de esas dos clases
- $\rightarrow$ es una relación de flujo, definida en una par de clases de seguridad. Para las clases *A* y *B*, $A \rightarrow B$ si y sólo si la información de la clase A tiene permiso para fluir a la clase B. La información fluye de *A* a *B* cuando la información asociada con la clase *A* afecta al valor de la información asociada con la clase *B*.

Requerimientos de seguridad:

Un modelo de flujo FM es seguro si y sólo si la ejecución de una secuencia de operaciones no puede dar lugar a un flujo que viola la relación $\rightarrow$, y forma un modelo de retículo bajo ciertas suposiciones. Ejemplos de retículos seguros son aquellos que están ordenados linealmente. El siguiente ejemplo constituye un retículo seguro:

1. Cada objeto es asignado a una clasificación (desclasificado, confidencial, secreto, alto secreto). Esta clasificación denota el nivel de seguridad del contenido de la información.

2. A cada sujeto se le asigna una acreditación para acceder a ciertas clases de información (Comandante, Coronel, Capitán). El término nivel de seguridad denota la acreditación de un sujeto o la clasificación de un objeto.
3. Cada objeto y cada sujeto pueden estar afiliados con un Compartimento (Marina, tierra, aire), que categoriza el contenido de la información para el objeto y representa la clasificación para el sujeto.
4. La clase seguridad SC de un objeto o sujeto, está especificada por el par (A,B), donde A es la acreditación o nivel de seguridad y B es el Compartimento, Ej: (Coronel, Marina), (Alto secreto, Tierra).

Un flujo de información seguro está definido por un conjunto de clases de seguridad parcialmente ordenado. Esto implica que el control de acceso sea gobernado por el orden de las clases de seguridad; un sujeto *S* tiene permiso de acceso a un objeto *O* sí y sólo sí se satisfacen dos condiciones:

1. *S* tiene acreditación apropiada relativa a la clase del objeto *O*.
2. El compartimento para *S* está contenido en el compartimento para *O*. Esta condición es una aplicación del mínimo privilegio.

Existe un mecanismo de certificación para verificar si el flujo de información es seguro dentro de un programa. Este mecanismo explota las propiedades de la estructura de retículo entre las clases.

Modelo de Bell y LaPadula

El modelo formalizado por Denning había sido construido con anterioridad por Bell y LaPadula, patrocinado por las Fuerzas Aéreas de los EE.UU. Este modelo constituye un núcleo de seguridad mediante un autómata finito, donde se definen las reglas que permiten la transición de un estado a otro. El modelo de Bell y LaPadula es la implementación del control de acceso mediante flujo de información más conocida.

Este modelo cuenta con los siguientes componentes:

- Consiste en un conjunto de sujetos, un conjunto de objetos, y una matriz de acceso Tiene varios niveles de seguridad ordenados. Cada sujeto tiene su acreditación y cada objeto tiene una clasificación, perteneciendo a un nivel de seguridad. Cada sujeto tiene un nivel de "acreditación actual" que no puede superar la acreditación del sujeto.
- Los sujetos pueden tener los siguientes permisos:
    - Sólo lectura
    - Escritura
    - Ejecución
    - Lectura y escritura

Además de estos accesos, el sujeto que crea un objeto tiene los atributos de control de ese objeto. Un sujeto puede pasar a otro sujeto cualquiera los permisos de acceso a cualquier objeto para el cual tiene control de los atributos. Sin embargo, el control de los atributos no puede ser pasado.

El modelo impone las siguientes restricciones en cuanto al flujo de información y control de acceso:

- Propiedad simple de seguridad. Un sujeto no puede tener permisos de lectura sobre objetos cuya clasificación es mayor que el nivel de acreditación del sujeto.
- Propiedad estrella. Un sujeto tiene permiso de escritura sólo a aquellos objetos cuya clasificación es mayor o igual que el nivel actual de acreditación del sujeto. Tiene acceso de lectura sólo a aquellos objetos cuya clasificación es menor o igual al actual nivel de acreditación y tienen permiso de lectura-escritura a aquellos objetos cuya clasificación es igual al nivel actual de acreditación.
- Los axiomas que gobiernan los accesos de los usuarios son los siguientes:
- Ningún usuario puede leer información clasificada por encima de su nivel de acreditación.
- Ningún usuario puede bajar el nivel de clasificación de la información

Bell y LaPadula modelaron este comportamiento mediante una máquina de estado finito. Se define el concepto de estado seguro y se consideran sólo las transiciones que llevan al sistema a un estado seguro.

El modelo de Bell y LaPadula tiene algunos inconvenientes: los niveles de seguridad de los objetos son estáticos y la propiedad estrella puede ser demasiado restrictiva en muchas aplicaciones. Por ejemplo, la propiedad estrella dictamina que un sujeto de un nivel no puede comunicarse con sujetos de niveles más bajos. En un sistema de computación un proceso a nivel i debe ser capaz de escribir información a procesos a niveles menores si la información no depende de objetos protegidos a nivel i o mayores.

El modelo de retículo puede clasificarse como un modelo de control de acceso no discrecional, pues existe una política claramente establecida que indica cuáles son los posibles accesos de un sujeto cualquiera, sin que éstos puedan ser modificados discrecionalmente. Esta política viene definida por las reglas establecidas.

4.4 Núcleo de seguridad

El mecanismo básico que controla la seguridad del sistema [Lan81] debe estar basado en un monitor de referencia que controla cualquier acceso entre sujetos y objetos. Es el que implementa el principio de mediación completa ya que controla todos los accesos. Se encarga de implementar el modelo de protección escogido para el sistema, sea cual sea éste, y forma el núcleo de la Base de Computación Fiable del sistema.

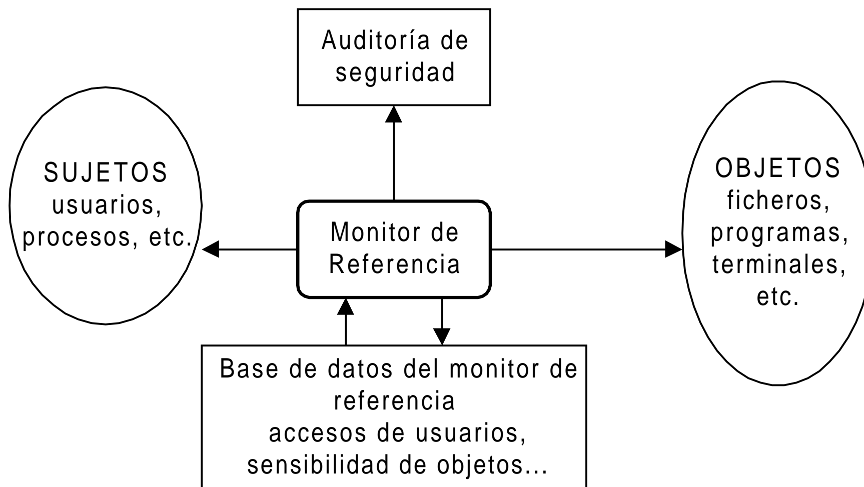


Figura 4.3: Monitor de referencia

El concepto de monitor de referencia fue adoptado por Lampson y define sujetos, objetos, una base de datos monitor de referencia y una auditoría de seguridad. La base de datos consiste en varios tipos de datos relativos a la seguridad, tales como acceso de los usuarios, susceptibilidad de los objetos, necesidad de conocimiento (*need-to-know*). Esta base de datos proporciona información al monitor de referencia sobre qué sujetos están autorizados a acceder a qué objetos. Esta base de datos puede considerarse que se ubica dentro del campo de la política de seguridad más que del control de acceso en sentido estricto.

El mecanismo de auditoría de seguridad permite a usuarios y gestores de seguridad monitorizar y recopilar información de una variedad de eventos que representan intentos de acceso, con éxito o no, con objeto de identificar intentos para saltar la barrera de protección.

El núcleo de seguridad debe situarse en las capas más bajas del sistema, y es el responsable de todas las operaciones relativas a la seguridad. y debe ser lo más pequeño posible, siguiendo el principio de ahorro de mecanismos.

4.5 Implementaciones y variantes de la matriz de control de acceso

En la matriz de control de acceso se colocan los objetos en las columnas y los sujetos en las filas, con los permisos de acceso situados en las entradas de la matriz. Esta matriz en general es muy dispersa, pues normalmente cada objeto sólo puede ser accedido por un número reducido de sujetos. Por otro lado hay que tener en cuenta que los sujetos y los objetos no necesariamente deben estar situados en la misma máquina.

Una implementación directa de la matriz de acceso sería muy ineficiente. La eficiencia podría ser mejorada descomponiendo la matriz de acceso en filas o en columnas, dando lugar a dos métodos ortogonales que se estudian a continuación, y que constituyen diferentes maneras de representación de la matriz de acceso: las listas de control de acceso, las capacidades y la representación mixta.

4.5.1 Listas de control de acceso

Consiste descomponer la matriz por columnas. A cada objeto se le asocia una **lista de control de acceso** (LCA): la lista de sujetos que pueden acceder al mismo (es decir, la columna correspondiente a dicho objeto, sin las celdas vacías), indicando además los permisos de acceso.

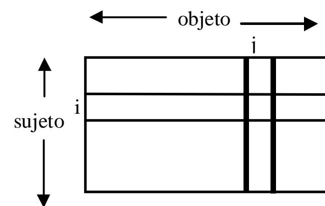


Figura 4.4: Matriz de acceso vista por columnas

Aunque los modelos han sido diseñados para sistemas más clásicos (paradigma procedural), en la siguiente figura se representan, siguiendo el método Unificado [BRJ96], un conjunto de objetos que utilizan sus listas de control de acceso cada vez que se efectúa una operación sobre ellos. En este caso se trata de un diseño orientado a objetos en el que todos los elementos son objetos.

Los objetos *A*, *B* y *X* efectúan concurrentemente operaciones sobre el objeto *I.* Antes de que *I* realice la operación, el mecanismo de control comprueba si el objeto (*A*, *B* o *X*) tienen acceso. En este caso se ha expresado haciendo que *I* busque en su lista de control de acceso, en función de si tiene o no permisos el objeto *I* devolverá el resultado o denegará la petición.

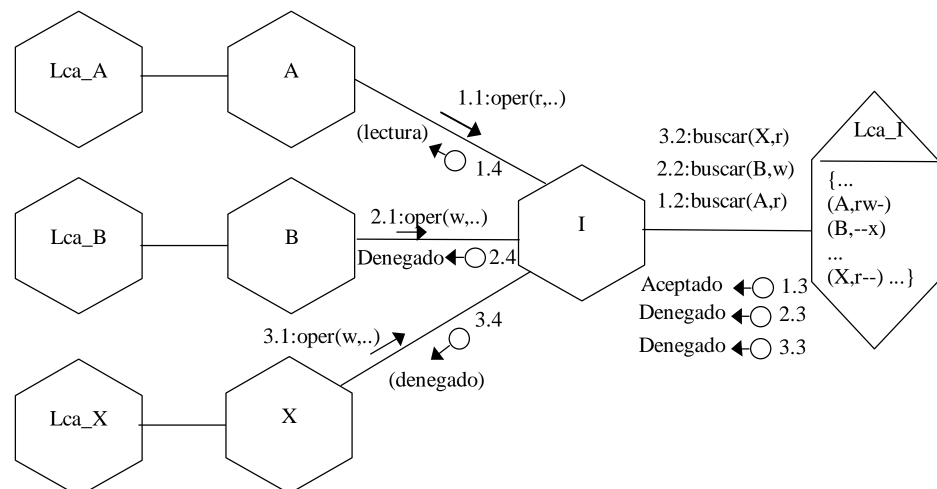


Figura 4.5: Funcionamiento del mecanismo de listas de control de acceso

4.5.1.1 Inconvenientes

- **Ineficiencia**. Es necesario consultar la lista de control de acceso cada vez que se pretende un acceso a una operación de un objeto, resultando considerablemente ineficiente. Cualquier sujeto puede intentar acceder a un objeto, aún sabiendo que no tiene permisos. El número de comprobaciones que deben realizarse es mayor. El funcionamiento puede verse como un sistema abierto en el que se puede hacer en principio cualquier operación, y entonces se comprueba siempre para ver si la operación es legal.
- **Dificultad de determinación de privilegios de un sujeto**. Es complicado determinar todos los permisos de acceso de un sujeto (objeto) dado, puesto que se encuentran repartidos por las listas de control de acceso de los objetos a los que puede acceder.

- **Conocimiento exhaustivo de sujetos**. Cada objeto debe ser conocedor de los sujetos existentes en el sistema que puedan acceder a él (para poder formar su lista de control de acceso). Esto conlleva una dependencia entre objeto/sujeto (o servidor/cliente), que puede resultar compleja en caso de sistemas distribuidos.

4.5.1.2 Ventajas

Las principales ventajas de las listas de control de acceso incluyen:

- **Fácil revocación**. La eliminación de permisos de acceso a sujetos que los poseían es muy simple, rápida y eficiente. Puede ser lograda simplemente eliminando al sujeto de la lista de control de acceso del objeto correspondiente.
- **Fácil revisión de un acceso**. Puede ser determinado fácilmente qué sujetos tienen permisos de acceso a un objeto examinando directamente la lista de control de acceso del objeto. Sin embargo es difícil determinar a qué objetos tiene acceso un sujeto.

4.5.1.3 Grupos de protección

En ocasiones se introduce el concepto de **grupo de protección** para reducir la sobrecarga de almacenamiento de las listas de control de acceso, y en consecuencia el tiempo de búsqueda en ellas. En este caso los sujetos se agrupan en dominios de protección y la lista de control de acceso consiste en nombres de grupos con sus respectivos permisos de acceso. El número de entradas de la lista de control de acceso está limitada al número de grupos de protección.

Todos los sujetos pertenecientes a un mismo grupo de protección dispondrán de los mismos permisos de acceso, restringiendo la discrecionalidad del acceso, que no puede llegar al nivel de detalle (grano fino) que se logra con una lista de control de acceso genérica.

Grupos de protección en el sistema operativo Unix

Un ejemplo de ello es el caso del sistema operativo Unix, en cuanto al modelo de protección que utiliza para el acceso a los ficheros [TW97]. Se definen así tres dominios diferentes: propietario, grupo y resto. Cada fichero dispone de una lista de control de acceso con una entrada para el par (propietario, permisos) otra para el par (grupo, permisos) y otra para la pareja (resto, permisos). Cada usuario tiene asignado en el sistema una serie de grupos a los que pertenece. En función del sujeto que realice la petición se buscará en la lista de control de acceso y se comprobarán los permisos. Si el usuario coincide con el propietario del fichero se comprueban los permisos del primer par, si pertenece al grupo indicado en la lista el segundo par, y finalmente se comprueba el tercer par (puede considerarse que todos los usuarios pertenecen al grupo “resto”).

4.5.1.4 Propagación de los permisos de acceso

Existen dos modos de propagación (modificación, concediendo permisos a otros sujetos) de los permisos de acceso, el control autónomo y el control jerarquizado, en función de las entidades autorizadas para modificar los permisos de acceso de un objeto.

Propagación con control autónomo

En una política de **control autónomo**, el propietario de un objeto tiene un permiso de acceso especial por el cual puede modificar la lista de control de acceso del objeto, eliminando o añadiendo nuevos permisos. Generalmente el propietario es el proceso creador del objeto, aunque esto depende del diseño del sistema operativo.

Propagación con control jerarquizado

En el **control jerarquizado**, cuando se crea un objeto su propietario especifica un conjunto de procesos que tendrán permisos para modificar la lista de control de acceso. Los procesos se disponen en una jerarquía de forma que un proceso puede modificar la lista de control de acceso asociada con todos los procesos que se encuentran por debajo en la jerarquía.

4.5.2 Capacidades

El método basado en **capacidades** [Gos91] consiste en descomponer la matriz de acceso por filas. Cada sujeto tiene una lista de los objetos a los que puede acceder, representados cada uno de ellos por una capacidad, indicando los permisos que el sujeto posee sobre dicho objeto. Se trata por tanto de una tupla (objeto, permisos). La simple posesión de una capacidad por parte de un sujeto concede el acceso al objeto al que se refiere.

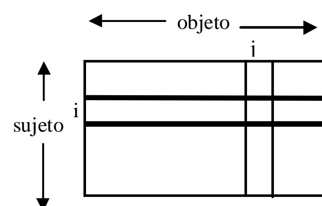


Figura 4.6: Matriz de acceso vista por filas

Dominios de protección

Es posible agrupar los sujetos, de forma que cada grupo posea un conjunto de capacidades. A este conjunto de capacidades se le denomina dominio de protección.

Implementación de las capacidades

Aunque existen diferentes modos de implementación de las capacidades, es bastante habitual que éstas estén formadas por al menos dos campos: un identificador del objeto y un campo de permisos de acceso. El campo con el identificador del objeto pueden servir también para la localización de dicho objeto, consiguiendo así que la capacidad pueda ser utilizada como mecanismo de direccionamiento del objeto además de como mecanismo de protección.

En la siguiente figura se muestra un ejemplo de forma análoga al apartado anterior, dentro del contexto del paradigma de la orientación a objeto, en el que el acceso a las operaciones sobre los objetos está gestionado a través de capacidades. En este caso se presentan las capacidades como elementos internos al objeto, aunque podría seguirse la misma filosofía del caso de las listas de control de acceso, creando un objeto “lista de capacidades”.

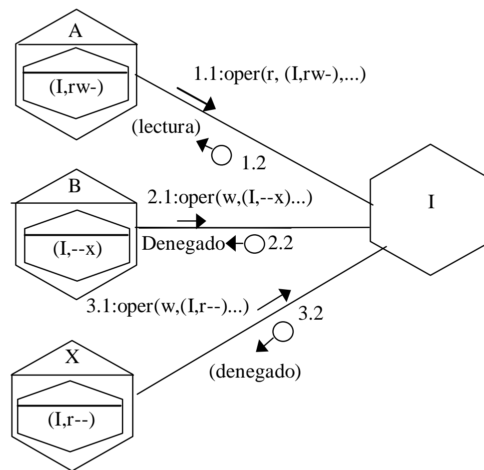


Figura 4.7: Funcionamiento del mecanismo de capacidades

4.5.2.1 Falsificación de capacidades

Uno de los problemas que presentan las capacidades es la necesidad de evitar que éstas puedan ser manipuladas o falsificadas. Para ello existen diferentes modos de implementación de las capacidades, que generalmente se agrupan en tres categorías [VVG95]: **etiquetadas** (*tagged*), **particionadas** o **segregadas** (*partitioned* o *segregated*) y **dispersas** (*sparse*). En el capítulo 10 se desarrollan cada una de estas variantes.

4.5.2.2 Ventajas

- **Simplicidad**. La validación del acceso es muy simple, pues la simple posesión de la capacidad con los permisos es suficiente para la concesión del acceso
- **Eficiencia**. No hay que hacer ninguna búsqueda en una lista en cada acceso, simplemente hay que comprobar los permisos. No se puede acceder arbitrariamente a cualquier objeto e intentar una operación, al contrario que en las listas de control de acceso, puesto que para poder intentarlo hay que tener previamente una capacidad para el objeto. El sistema puede verse como un sistema cerrado en el que se van añadiendo sólo las operaciones necesarias creando las capacidades adecuadas. Esta es una aproximación más prudente en la construcción de sistemas seguros.

4.5.2.3 Inconvenientes

- **Revocación y revisión de los permisos de acceso**. No existe un registro de la localización de las capacidades sobre un determinado objeto. Los sujetos que las poseen pueden a su vez enviárselas a otros propagando éstas cuanto quieran. Así pues, no es fácil en principio la revocación, ni la revisión de ciertos permisos de acceso en una capacidad. Puede solucionarse esto utilizando fachadas, es decir, objetos intermedios que se crean específicamente para cada cliente.
- **Control de propagación.** Otro inconveniente es la manera de controlar la cesión de permisos. En principio el sujeto que posee una capacidad puede realizar una copia y enviar ésta a quien desee propagando de esta manera la capacidad por el sistema. Existen algunas técnicas que también permiten solucionar este problema en diferentes medidas.

  Estos y otros aspectos se tratarán con mayor detalle en capítulos posteriores.

4.5.3 Mixta (Clave-permisos)

Se mantienen por separado, para eliminar inconvenientes de los métodos anteriores, las capacidades y los permisos. Cada sujeto tiene una lista de capacidades (objeto, clave) y cada objeto una lista de control de acceso (clave, permisos). De esta manera, sujetos que tengan los mismos permisos, pueden compartir la misma clave y para cambiar los permisos, simplemente se cambian los permisos asociados a la clave.

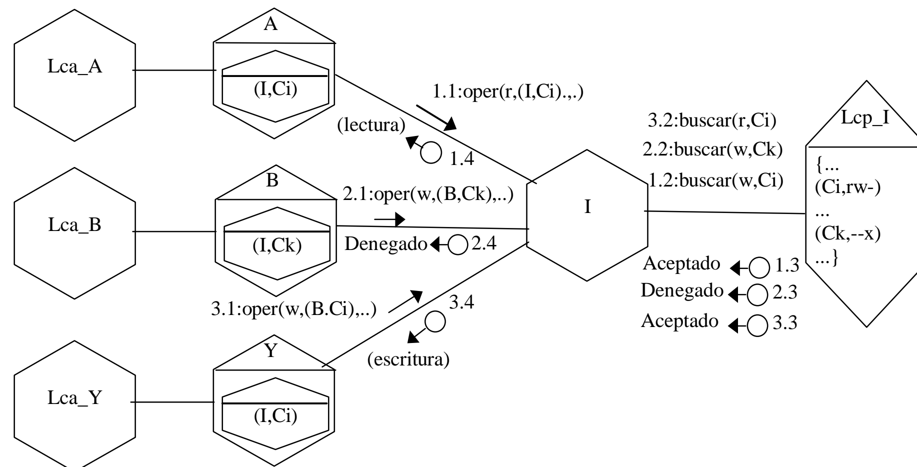


Figura 4.8: Funcionamiento del mecanismo mixto clave-capacidad

En el caso de que existan tantas claves como objetos pueden acceder a uno dado, se podrá implementar la revocación al nivel de cada objeto. Como inconveniente principal de este modelo es el mismo que tiene el modelo de las listas de control de acceso, es decir, que por cada invocación de una operación es necesario comprobar que el objeto que invoca tiene permisos.

5 Resumen de la Arquitectura de un Sistema Integral Orientado a Objetos

A continuación se enumeran los requisitos que debe cumplir un sistema integral orientado a objetos y se describe una arquitectura que permite obtener estos objetivos, que ya se habían enumerado con más brevedad en el capítulo 2.

Esta arquitectura [ATA+97] integra diferentes tendencias que, en general, se han aplicado para solucionar problemas concretos en sistemas determinados. Sin embargo, la arquitectura aquí expuesta 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 de uniformidad del sistema.

5.1 Requisitos de un 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*.

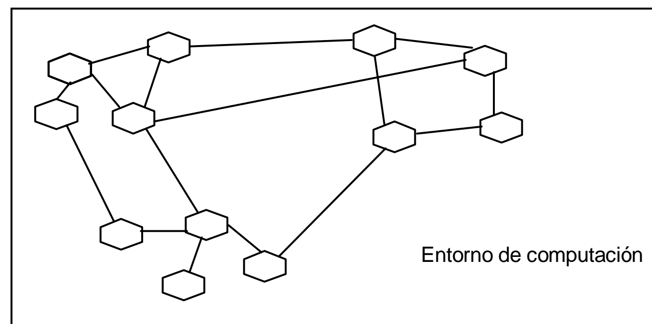


Figura 5.1: Entorno de computación compuesto por un conjunto de objetos homogéneos

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 [Álv98]:

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

5.1.1 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].

5.1.2 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[1] a otros objetos. Toda la semántica de un objeto se encuentra encapsulada dentro del mismo.

5.1.3 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.

5.1.4 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.

[1] Se utilizarán indistintamente las expresiones envío de mensaje, llamada o invocación de método y llamada o invocación de operación.

5.1.5 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.

5.1.6 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.

5.1.7 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.

5.1.8 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.

5.1.9 Flexibilidad

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.

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

La estructura consiste en una máquina abstracta orientada a objetos 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 de manera transparente.

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.

5.2.1 Máquina Abstracta Orientada a Objetos

Dotada de una arquitectura reflectiva, proporciona un modelo único de objetos para el sistema.

5.2.1.1 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.

5.2.1.2 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.

5.2.2 Sistema Operativo Orientado a Objetos

Estará formado por un conjunto de objetos que proporcionen funcionalidad que en otros entornos se considera parte del sistema operativo.

5.2.2.1 Transparencia: persistencia, distribución, concurrencia y protección

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

5.2.3 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.

5.2.4 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.

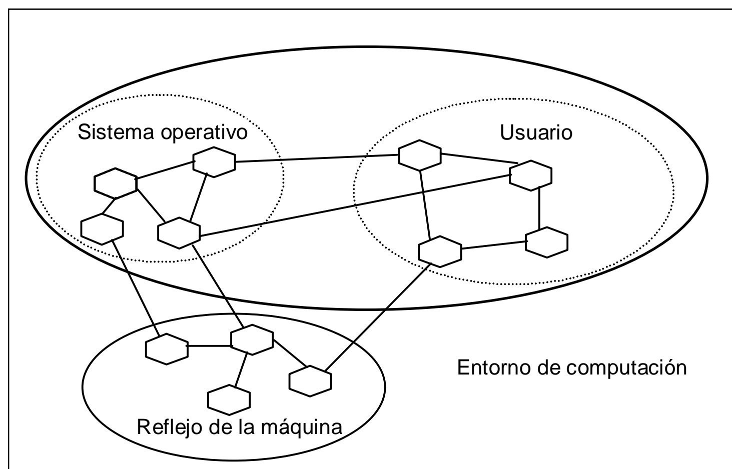


Figura 5.2: Espacio único de objetos homogéneos

5.2.5 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 la localización de un objeto al resto del sistema.

Figura 5.3: Propiedades incorporadas en la arquitectura del sistema integral y relaciones con los objetivos del mismo

5.3 Modelo único de objetos del Sistema Integral

El modelo de objetos que ha sido definido para el sistema sigue, intencionadamente, las especificaciones de los modelos de objeto de las metodologías de orientación a objetos más populares, como la de Booch [Boo91, 94]. El objetivo es aprovechar las ventajas de los conceptos totalmente establecidos, así como soportar todas la semántica del modelo de objetos que se usa desde la fase de análisis y diseño hasta la fase de implementación.

No todas estas características tienen que estar implementadas por el mismo elemento del sistema integral. De hecho, algunas de ellas, como la persistencia, es más interesante introducirlas de manera que (aunque transparente) su uso pueda ser opcional. Por tanto, se pueden dividir las características en grupos en función del elemento del sistema que se ocupe de implementarlas.

5.3.1 Parte del modelo en la máquina abstracta

La máquina abstracta 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.

- **Identidad única de los objetos**, basada en el uso de referencias.
- **Encapsulación**. Acceso a los objetos únicamente a través de sus métodos.
- **Clases**. Utilizadas para derivar tipos.
- **Relaciones de herencia múltiple** (es-un).
- **Relaciones de agregación** (es-parte-de).
- **Relaciones generales de asociación.**
- **Polimorfismo y comprobación de tipos en tiempo de ejecución.**
- **Manejo de excepciones.**

5.3.2 Parte del modelo en el sistema operativo

Se encargará de conseguir las propiedades que más comúnmente se asocian con funcionalidad propia de los sistemas operativos:

- **Concurrencia**
- **Persistencia**
- **Distribución**
- **Protección**

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.

Sin embargo, en el diseño del sistema operativo, se prevé que pueda ser necesario por determinadas razones, incluir elementos propios de funcionalidad anterior como parte de la máquina abstracta (como será en el caso de la protección), o al menos repartir la misma entre los objetos del SO y responsabilidades introducidas en la máquina.

5.4 Arquitectura de referencia de la máquina abstracta

En este apartado se describe una arquitectura de referencia de una máquina abstracta orientada a objetos con las propiedades necesarias para dar soporte a un sistema integral orientado a objetos.

5.4.1 Propiedades fundamentales de una máquina abstracta para un SIOO

Las características fundamentales de la máquina se muestran a continuación:

- Modelo único de objetos, que implementará la máquina
- Identificador único de objetos
- Uniformidad en la OO, único nivel de abstracción
- Interfaz de alto nivel, independiente de estructuras de bajo nivel
- Juego reducido de instrucciones
- Flexibilidad (extensión)

5.4.2 Principios de diseño de la máquina

A continuación se describen brevemente los principios de diseño de la máquina abstracta [DAG99]:

- **Elevación del nivel de abstracción**, con una interfaz de instrucciones de alto nivel orientadas a objetos. Se eleva el nivel de forma homogénea a un nivel de orientación a objetos.
- **Conjunto reducido de instrucciones**. Contrariamente a lo que ocurre en la máquina virtual de Java [LY97], y más en la línea del espíritu inicial de la máquina Smalltalk [GR83], el conjunto de instrucciones debe reducirse al mínimo necesario para expresar los conceptos de la orientación a objetos. El conjunto de instrucciones proporcionará instrucciones declarativas que describen clases, referencias agregadas, y referencias asociadas; e instrucciones de comportamiento para el cuerpo de los métodos.
- **Sin mezcla de niveles de abstracción**. Existirá sólo un tipo de objetos, sin tipos primitivos especiales como en Java. La interfaz no utilizará estructuras de implementación de bajo nivel como una pila.

5.4.3 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 implementación 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.
- **Jerarquía de clases básicas.** Existirán una serie de clases básicas en el sistema que siempre estarán disponibles, siendo la raíz de la jerarquía una clase denominada comúnmente Object. Serán las clases básicas del modelo único de objetos del sistema. Cada distribución del sistema establecerá los mecanismos necesarios para proporcionar la existencia de estas clases básicas. Normalmente serán implementadas directamente de manera primitiva por la propia máquina, pero en ciertos casos podrían estar implementadas con el lenguaje de programación del usuario.

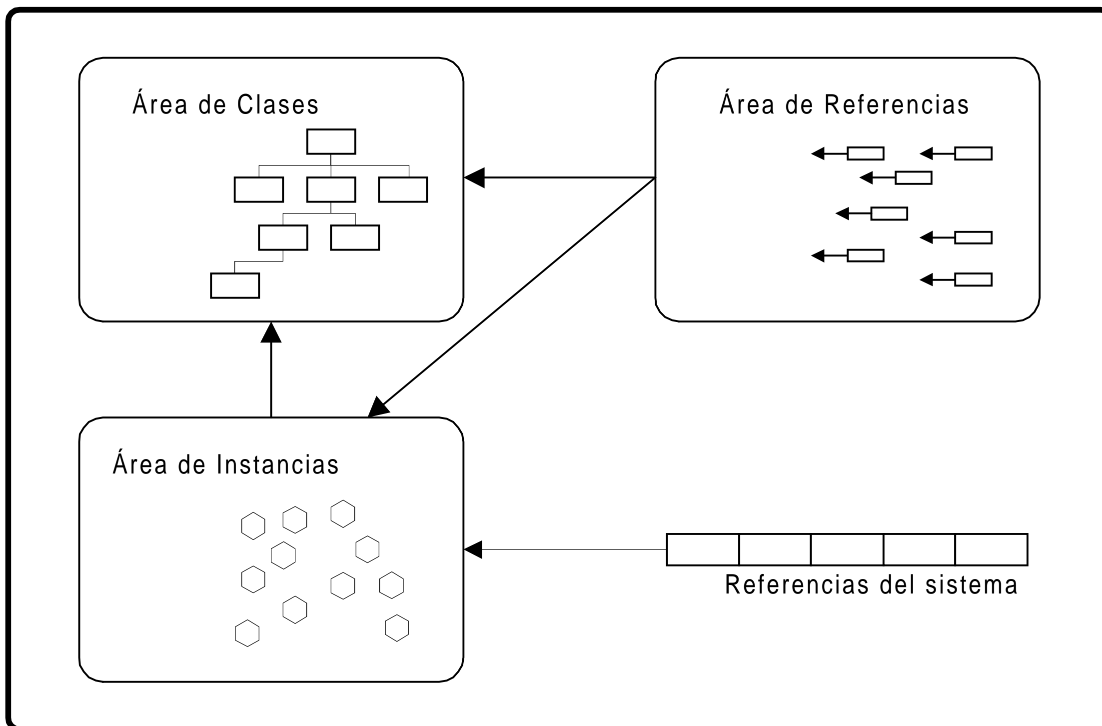


Figura 5.4: Estructura de referencia de una máquina abstracta para el sistema integral

5.4.4 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. La unidad de descripción en un SIOO son las clases. Por tanto el lenguaje ensamblador permitirá describir una clase mediante las instrucciones:

5.4.4.1 Instrucciones declarativas

Esta arquitectura almacena la descripción de las clases, así que puede considerarse que existen instrucciones cuyo resultado es la descripción de la información de una clase:

- **Nombre de la clase**
- **Relaciones de herencia**
- **Relaciones de agregación**
- **Relaciones de asociación**
- **Métodos de la clase**, con los parámetros y referencias locales que utilicen.

5.4.4.2 Instrucciones de comportamiento

Que permitan definir el comportamiento de la clase. Es decir, el comportamiento de sus métodos. Describirán el código que compone el cuerpo de cada método:

- **Invocación de método** a través de una referencia. Son en esencia la única operación que se puede realizar en un método.
- **Retorno de un método**.
- **Control de flujo**. Instrucciones para controlar el flujo de ejecución de un método: saltos, saltos condicionales, etc.
- **Excepciones**. Instrucciones de control de flujo para gestionar las excepciones
- **Instrucciones para trabajar con los objetos** a través de las referencias:
  - **Creación y borrado de objetos** a través de una referencia.
- Etc.

5.4.5 Ventajas del uso de una máquina abstracta

Las ventajas del uso de una máquina abstracta como ésta 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.

5.5 Extensión de la máquina abstracta para proporcionar la funcionalidad del sistema operativo. Reflectividad

En general, la funcionalidad asignada al sistema operativo será proporcionada mediante objetos normales de usuario. Sin embargo, en algunos casos será necesario modificar de ciertas maneras la máquina, incluyendo el soporte que sea necesario para la implementación de esta funcionalidad. En general existen tres maneras de introducir esta funcionalidad en el sistema, que podrán usarse por separado o, lo que es más probable, en combinación. Además, casi siempre será junto con objetos normales de usuario que proporcionen la funcionalidad restante.

5.5.1 Modificación interna de la máquina.

Se trata de realizar cambios internos en la máquina para incorporar la funcionalidad que se necesite. Estos cambios involucrarán una modificación en la semántica de las operaciones internas, o bien un cambio en la interfaz de la máquina (como por ejemplo la adición de una nueva instrucción).

5.5.2 Extensión de la funcionalidad de las clases básicas

En este caso se modifica la funcionalidad que proporcionan las clases básicas en el sistema. Se pueden añadir atributos y métodos, y también cambiar la semántica de métodos existentes (aunque no debería ser habitual). Por ejemplo, al añadirse un nuevo método a la clase raíz de la jerarquía de clases del sistema (por ejemplo devolverNombre), automáticamente todos los objetos pasan a tener ese método a su disposición.

5.5.3 Colaboración con el funcionamiento de la máquina. Reflectividad.

Permitir que objetos del sistema operativo (objetos normales) colaboren con la máquina cuando esta no pueda realizar una tarea por sí misma, mediante una arquitectura reflectiva [Mae87]. La máquina tendrá una arquitectura en la que los objetos componentes de la máquina se expongan como si fueran objetos normales.

De esta manera, los objetos de la máquina pueden usarse como objetos normales, y a su vez pueden usar estos objetos. Cuando un objeto de la máquina no puede continuar por alguna razón, llama a un objeto del sistema operativo (mediante una excepción o por llamada directa). Este objeto colaborará con la máquina y posiblemente con otros objetos para resolver en conjunto el problema. El sistema puede verse como un conjunto de objetos que interactúan, todos con las mismas características. Algunos de ellos serán objetos de la máquina, otros serán objetos normales de usuario y algunos de estos últimos implementarán funcionalidad del sistema operativo. Sin embargo, el origen de un objeto es transparente para todos los demás, e incluso para sí mismo.

Como ejemplo, la persistencia puede ser proporcionada con objetos de la máquina colaborando con objetos del sistema operativo para guardar y recuperar instancias en el disco. Así, la máquina, cuando en su funcionamiento normal se haga referencia a un objeto que no se encuentra en el área de instancias, llamará de manera reflectiva a un objeto de usuario (gestor de persistencia). Este objeto se ocupará de recuperar del almacenamiento persistente a este objeto y llamar a un objeto de la máquina (área de instancias) para colocar su estado en el área de instancias. El efecto de esta colaboración es que la persistencia de los objetos es transparente para las aplicaciones. Al colaborar reflectivamente con objetos de usuario, se puede cambiar dinámicamente la implementación de estos, como objetos normales de usuario que son. Se pueden desarrollar fácilmente, pues, mejores implementaciones del gestor de persistencia, o nuevos gestores de persistencia especializados que aporten funcionalidad

adicional (doble comprobación de escritura, gestión de transacciones, encriptación de objetos, etc.). Sin embargo, el núcleo del sistema no necesita modificaciones de ningún tipo (recompilaciones del núcleo).

La reflectividad está reconocida como una técnica importante para lograr flexibilidad (extensibilidad, adaptabilidad, etc.) en los núcleos de los sistemas operativos [Cah96, GW96], y además, en este caso contribuye a lograr la uniformidad en la orientación a objetos en la extensión del sistema.

6 Requisitos de un Mecanismo Básico de Protección para un Sistema Integral Orientado a Objetos

6.1 Mecanismo de protección para un Sistema Integral Orientado a objetos

Dentro de todo el ámbito de la seguridad en un sistema, el contexto concreto en el que se desarrolla esta tesis se define a continuación: se trata de *definir un mecanismo de protección que controle el uso de recursos y servicios (objetos) en un sistema integral orientado a objetos*.

6.1.1 Plataforma de desarrollo del mecanismo de protección

Como base para el estudio de la seguridad y el desarrollo de un nuevo mecanismo de protección se utiliza un Sistema Integral Orientado a Objetos, puesto que es este tipo de sistemas al que se tiende en estos momentos y donde surgen los mayores agujeros de seguridad relativos a la interoperabilidad de objetos. En el siguiente apartado se explican las características básicas del Sistema Integral Orientado a Objetos elegido como plataforma.

6.1.2 Seguridad en el uso de recursos y servicios

Como se ha descrito en un capítulo anterior sobre conceptos de seguridad, la seguridad abarca tres aspectos dentro del sistema: seguridad en el uso de recursos y servicios, seguridad en el acceso al sistema, y seguridad en el uso de redes.

El desarrollo de esta tesis se centra en el aspecto de seguridad en el uso de recursos y servicios, que como se había visto, que ha ampliado su ámbito de influencia con la proliferación de las tecnologías de objetos y la distribución.

En un Sistema Integral Orientado a Objetos (SIOO), el único recurso a proteger es el objeto, puesto que es la única abstracción existente en el sistema. Todos los objetos son de la misma categoría, sin establecer diferencias entre ellos. Esto hace más sencillo conceptualmente al mecanismo de protección.

6.1.3 Mecanismo de protección y política de seguridad

El mecanismo de protección que se pretende desarrollar supone un núcleo de seguridad para el sistema. Un buen mecanismo de protección deberá ser lo suficientemente flexible para que puedan implementarse sobre él diversas políticas de seguridad en función del tipo de sistema sobre el que se desee trabajar.

Aunque en esta tesis se muestran varias políticas de seguridad, el núcleo principal es el desarrollo de un mecanismo de protección. Las políticas que se presentan que sirven como demostración de la flexibilidad y posibilidades del mecanismo.

6.2 Requisitos de protección

A continuación se establecen los requisitos 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.

6.2.1 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.

Diferentes tipos de sistemas necesitarán diferentes políticas de seguridad. Por tanto, es deseable que el mecanismo de protección, situado en las capas más bajas del sistema sea lo suficientemente flexible para llevar a cabo cualquier política.

Sistemas empotrados

Como ejemplo de los diferentes entornos en los que se puede utilizar un sistema integral orientado a objetos, con diferentes necesidades en cuanto a seguridad están por ejemplo los sistemas empotrados [Álv98]. Los **sistemas empotrados** (*embedded systems*) se caracterizan por ser sistemas de computación destinados a una tarea específica, normalmente con unos recursos limitados, como la memoria disponible. Se llaman empotrados porque están integrados físicamente con el elemento con el que trabajan. El hardware sobre el que funcionan suele ser muy variado, y depende del destino de cada sistema. Ejemplos de sistemas empotrados son sistemas de control de maquinaria industrial, controladores de motores, etc.

En este tipo de sistemas la protección no se necesita para proteger los recursos del sistema frente a usuarios malintencionados o programas de autor desconocido, ni se necesitan políticas de seguridad complejas. Esto es así ya que en esos sistemas no existen múltiples usuarios, ni acceso exterior al sistema y todos los programas sueles estar escritos específicamente para el sistema. En este caso se necesita simplemente un mecanismo de protección que permita controlar errores de programación en el sistema, evitando que funciones mal programadas hagan llamadas a operaciones que no deberían utilizar, etc.

Es deseable que un mecanismo de protección permita soportar las necesidades elementales de este tipo de sistemas. Tampoco es posible utilizar un mecanismo de seguridad complejo del que sólo se utilice una parte pequeña para usar en estos sistemas. La conservación de memoria en estos sistemas es muy importante, y no se puede incluir funcionalidad que ocupe espacio y no sea necesario utilizar, simplemente porque forma parte indisoluble del sistema de seguridad del que se usa una pequeña porción. El mecanismo de seguridad debe ser suficientemente flexible como para poder incluir únicamente la parte que verdaderamente se utilice.

6.2.2 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 explicaba 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.

6.2.3 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.

6.2.4 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.

Esto puede considerarse un aspecto más de la uniformidad, ya que no se hace distinción entre objetos para la protección en función de su tamaño.

6.2.5 Protección de granularidad fina

Por otro lado, la protección debe extenderse al nivel de los métodos individuales de los objetos. Es decir, debe poderse especificar 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 completa, 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, en la línea del principio del mínimo privilegio.

6.3 Cumplimiento de los principios básicos de diseño de mecanismos

En el capítulo 3 sobre conceptos generales de seguridad, se describieron los principios básicos de diseño de un sistema de protección definidos por Saltzer y Schroeder [SS75]. Resulta necesario que el mecanismo de el mecanismo de protección que se diseñe cumpla también estos principios básicos, para así asegurar su efectividad.

Se recuerda aquí, pues, cuáles son estos principios básicos:

- **Mínimo privilegio**. Cualquier programa debe obtener únicamente los permisos sobre otros recursos que sean estrictamente necesarios para su funcionamiento, y nunca más que éstos.
- **Ahorro de mecanismos**. El mecanismo de protección debe ser lo más simple que sea posible, para facilitar su comprensión y reducir los riesgos de fallos en la implementación.
- **Aceptación**. El mecanismo debe inferir mínimamente en el trabajo del usuario para facilitar la aceptación por parte de éste, y por tanto, su utilización
- **Mediación total**. Es fundamental que la protección de los recursos se efectúe en todos y cada uno de los accesos. Es preciso que sea imposible esquivar el mecanismo por parte de usuarios malintencionados.
- **Diseño abierto**. El diseño del mecanismo no debe estar supeditado a su ocultación. Por el contrario, debe ser público, pudiendo así inspirar una mayor confianza.

En el apartado 3.9 se realiza una descripción más detallada sobre estos principios.

7 Panorama de Seguridad en algunos Sistemas Operativos

Como ya se ha podido apreciar en el repaso a la evolución histórica del tratamiento de la protección, los mecanismos utilizados en sistemas operativos clásicos, como la protección de memoria a través de los espacios de direcciones, la protección de ficheros y la división de operaciones entre modo supervisor/usuario, resultan insuficientes. Las nuevas necesidades de protección que surgen como consecuencia de la utilización de mecanismos cliente/servidor que facilitan la intercomunicación, y la expansión de los sistemas distribuidos y las tecnologías de objetos que explotan al máximo la interoperabilidad de objetos distribuidos necesitan mecanismos de protección más flexibles y uniformes.

A continuación se realiza una descripción de una selección de Sistemas Operativos significativos que se han desarrollado en los últimos años e incluso remontándose a épocas anteriores, y que proporcionan un mecanismo de protección diferente en mayor o menor medida a los métodos tradicionales de protección.

7.1 Sistema Operativo Amadeus

7.1.1 Características generales de Amadeus

Se desarrolló en el Departamento de Computer Science del Trinity College de Dublin, Irlanda, [CBH+92]. Se trata de un Sistema Operativo distribuido, de propósito general, que da soporte a un entorno de programación orientada a objetos. Pretende integrar soporte para bases de datos orientadas a objetos, y soporte para equilibrio de carga y paralelismo.

Utiliza un modelo de objetos pasivos, accedidos a través de procesos distribuidos. El sistema es capaz de ejecutar diferentes procesos propiedad de múltiples usuarios, que comparten los objetos de una forma segura.

Se trata, por tanto, de un sistema operativo moderno que incorpora la gestión de interoperabilidad entre objetos situados en espacios de direcciones diferentes. Para ello, será necesaria la incorporación de un mecanismo de protección que se muestra a continuación.

7.1.2 Mecanismo de protección

El mecanismo está basado en **listas de control de acceso** y **aislamiento de objetos no confiables**, mediante la utilización de **ámbitos**.

Un ámbito es un grupo de objetos que pertenecen a un usuario común. Cada usuario puede poseer múltiples ámbitos, pero cada objeto pertenece a un único ámbito. Este mecanismo de protección permite que objetos que no son confiables se aíslen en ámbitos separados.

Un ámbito está formado por un conjunto de contextos, donde se mapean estos objetos. Por tanto es imposible que un objeto de un ámbito interfiera sobre un objeto de un ámbito diferente. Los objetos de ámbitos diferentes están separados por límites de protección hardware al estilo tradicional. Un ámbito, por tanto, viene a ser como un espacio de direcciones donde se ejecutan un conjunto de objetos. Sólo las operaciones realizadas fuera de éste serán protegidas por el mecanismo basado en listas de control de acceso.

No existe, sin embargo, protección entre objetos del mismo ámbito. La protección se utiliza cuando desde un objeto se pretende llamar a otro de ámbito diferente. Entonces se realiza una invocación de cruce de ámbito a través del núcleo de Amadeus. El núcleo consulta la lista de control de acceso asociada al objeto llamado y se produce un cambio de ámbito, y desde allí se puede invocar cualquier objeto del nuevo ámbito sin necesidad de protección.

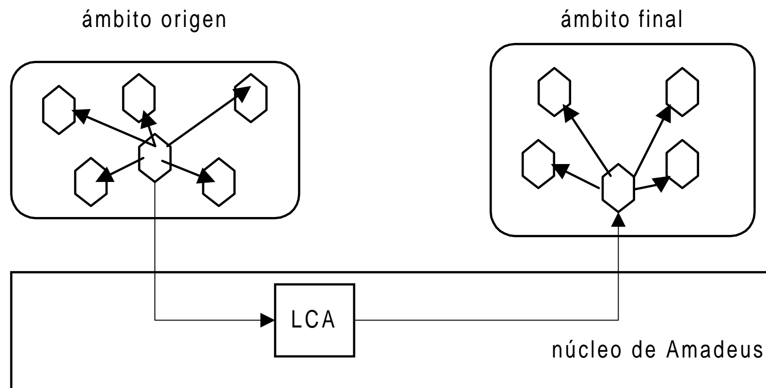


Figura 7.1: Mecanismo de protección en Amadeus

La autorización puede basarse en la identidad del usuario que llama o bien en el ámbito fuente (identificador de usuario efectivo). Se pretende asegurar que sólo los usuarios autorizados puedan invocar operaciones de un objeto dado. El control de acceso que se permite es al nivel de operaciones individuales de los objetos

Este sistema resulta muy similar al sistema operativo Guide, implementado por el INRIA de la Universidad de Grenoble, en Francia, y del que posteriormente se darán más detalles.

7.1.3 Crítica del mecanismo de protección

Inflexibilidad de aplicación a diferentes sistemas

El mecanismo de protección de este sistema está pensado para trabajar un entorno de usuarios. De hecho uno de los conceptos en los que se apoya es en el concepto de usuario, sobre el que se definen los ámbitos. La introducción del concepto de usuario dentro del mecanismo de protección obliga a que el sistema esté siempre basado en usuarios, y elimina la posibilidad de aplicación del mecanismo a sistemas de otros tipos.

Falta de protección intra-ámbito (sin principio de mínimo privilegio)

Dentro del ámbito se consideran los objetos seguros y no se establece protección. Esto va en contra del principio de mínimo privilegio puesto que están permitidos todos los accesos aún cuando no sean necesarios.

La existencia de ámbitos en los que no hay comprobación de protección dentro del mismo, resulta cuando menos limitado, sin que puedan evitarse errores de programación que el programador del ámbito pueda cometer inconscientemente, invocando operaciones que no debe aunque estén dentro del ámbito.

Mediación no total

El uso de ámbitos restringe el acceso a ciertas operaciones, evitando así la comprobación de permisos en todos y cada uno de los casos. Este hecho viola el principio general de mediación total por el que cualquier acceso debe ser comprobado previamente.

Escalabilidad limitada

La utilización de listas de control de acceso limita considerablemente la escalabilidad. El núcleo del sistema operativo se encarga de gestionar las listas de control de acceso de todos los ámbitos del sistema, lo cual puede resultar muy complejo con un aumento importante en el número de ámbitos.

Falta de protección intra-ámbito

La existencia de ámbitos en la que no hay comprobación de protección dentro del mismo, resulta cuando menos limitado, sin que puedan evitarse errores de programación que el programador del ámbito pueda cometer inconscientemente, invocando operaciones dentro del ámbito que no debe.

Falta de uniformidad

El mecanismo de protección no es uniforme. El acceso a las funciones del sistema operativo no está controlado siguiendo el mecanismo propuesto. Por otra parte no todos los accesos a operaciones de los objetos están protegidos, sólo aquellas que producen cambio de ámbitos. Además, no se trata a todos los objetos de igual forma, desde el momento en que realiza la protección a dos niveles: los que están dentro del ámbito no se protegen y los que están fuera sí.

7.2 Sistema Operativo Amoeba

7.2.1 Características generales del sistema

El Sistema Operativo Amoeba [MRT+90] es quizás uno de los más emblemáticos que se han construido en el entorno de los sistemas distribuidos, por ser uno de los pioneros, y por estar bajo la dirección de A.S. Tanenbaum, de la Universidad de Vrije, en Amsterdam, personaje ampliamente conocido en el mundo de la investigación y también en el de la educación a través de sus cuidados libros de Sistemas Operativos.

Amoeba es un Sistema Operativo distribuido basado en el paradigma de cliente-servidor. Su arquitectura se basa en un núcleo pequeño que se ejecuta en cada máquina y un conjunto de servicios que se ejecutan de forma independiente, y que pueden encontrarse en esa u otra máquina del sistema. Los servicios son controlados por procesos servidores que gestionan objetos de un determinado tipo. Los procesos clientes realizan operaciones sobre los objetos gestionados por procesos servidores. El servidor de bloques, servidor de ficheros plano, servidor de directorios, etc. son ejemplos de servicios proporcionados de forma independiente al núcleo. Existen algunos servicios manejados por el propio núcleo como el servicio de memoria.

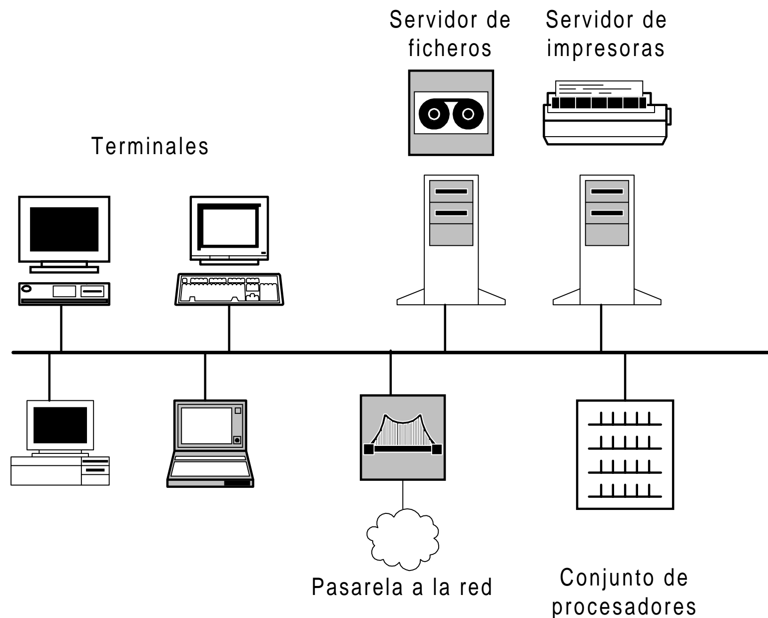


Figura 7.2: Arquitectura de Amoeba

7.2.2 Mecanismo de protección

El mecanismo de protección [MT86] de Amoeba utiliza dos niveles de protección: puertos para el acceso a los servidores y capacidades para el acceso los objetos individuales.

7.2.2.1 Capacidades y puertos

Cuando un proceso desea realizar una operación sobre un determinado objeto de un servidor, deberá disponer de una capacidad para él. El proceso envía la capacidad como parámetro del mensaje, el mensaje pasa por el núcleo, el cual extrae el nombre del puerto situado dentro de la capacidad. El puerto es la dirección del servidor que gestiona el objeto. El núcleo envía entonces el mensaje al puerto correspondiente llegando éste al servidor que gestiona el objeto. Este servidor realiza la comprobación de permisos sobre la capacidad recibida, y, en caso de tenerlos, invoca al objeto para que efectúe la operación y posteriormente envía los resultados al cliente.

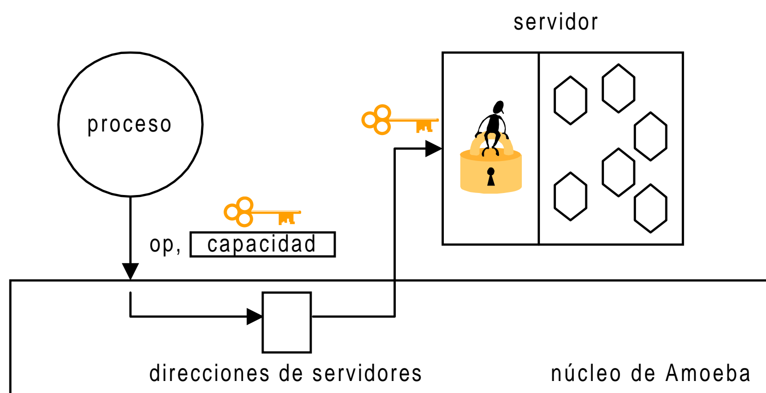


Figura 7.3: Control de acceso en Amoeba

7.2.2.2 Estructura de las capacidades

Las capacidades de Amoeba son un conjunto de bits almacenados en el propio espacio de direcciones del usuario. Disponen de los siguientes **campos**:

- El **puerto del servidor** que constituye la dirección del proceso servidor que gestiona el objeto, y será utilizada por el núcleo para localizarlo.
- A continuación aparece el **número del objeto**. Se trata de un número que identifica el objeto dentro del conjunto de objetos gestionados por un servidor.
- Seguidamente aparecen los **permisos**, que es un campo de 8 bits que indica las operaciones permitidas al poseedor del objeto. Así pues, el número máximo de operaciones a proteger en un objeto es 8.
- Por último un **campo de verificación** que se utiliza para dar validez a la capacidad, evitando que el usuario pueda falsificar o modificar la capacidad. Cuando la capacidad llega al servidor efectúa una comprobación de que los permisos no están falsificados mediante una función y un campo que guarda con la información del objeto. Tras la aplicación de dicha función el resultado debe ser el mismo que aparece en el campo de verificación.

Núm. Bits 48	24	8	48
Puerto Servidor	Objeto	Permisos	Campo de comprobación

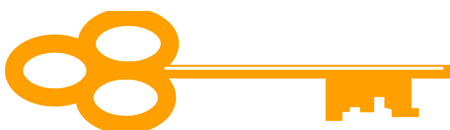


Figura 7.4: Capacidad en Amoeba

7.2.2.3 Modo de trabajo: capacidades de usuario encriptadas

Cuando un cliente pide la creación de un objeto a un servidor éste crea un número aleatorio y lo guarda en la tabla de objetos que tiene para gestionar todos los objetos que dependen de él. Este número aleatorio se va a utilizar como clave de encriptación y desencriptación. En el campo de verificación el servidor coloca una clave que será función de la clave aleatoria generada y también de los permisos concedidos.

De este modo se evita la posibilidad de que el cliente falsifique la capacidad, añadiendo por ejemplo más permisos de los que le han dado. Cuando el cliente hace una petición al servidor,

presenta la capacidad. El servidor busca la clave que tiene almacenada en la entrada de la tabla de objetos correspondiente al objeto en cuestión y aplica la función de encriptación a dicha clave y los permisos que recibe en el campo correspondiente de la capacidad. El resultado de esta función debe coincidir con el campo de validación de la capacidad. De lo contrario será porque el cliente ha modificado los permisos malintencionadamente, con lo que no se le permitirá el acceso.

La función de encriptación utilizada es una función de las llamadas "de sentido único", de forma que sea imposible que el cliente averigüe la clave de encriptación a través de los permisos que posee y el campo de validación.

Podemos concluir, por tanto, que Amoeba utiliza un mecanismo de protección basado en capacidades almacenadas en el espacio del usuario y protegidas criptográficamente del uso malintencionado por parte de éste.

7.2.2.4 Política de seguridad: servicio de denominación y de directorios

Existe un servicio de denominación, unido al servicio de directorios que proporciona una asociación entre el nombre simbólico de un objeto cualquiera y una o varias capacidades para él, con permisos diferentes dependiendo de quién pida la capacidad. Previamente el servidor ha debido crear las capacidades sobre sus objetos y las ha enviado al servicio de directorios. En este servicio se implementa por tanto la política de seguridad, manteniendo en este caso una entrada por cada objeto con las capacidades para el propietario, el grupo y el resto. El servicio de directorios es por tanto un gestor de una lista de control de acceso, en la que para cada objeto se guardan las capacidades para los distintos usuarios.

Las capacidades en Amoeba sirven para denominación, a través del puerto y el número del objeto, localización, a través del puerto, y protección a través de los permisos y el campo de validación.

El mecanismo de protección está localizado fuera del núcleo del sistema operativo, y se mantiene distribuido en cada uno de los procesos servidores.

7.2.3 Crítica del mecanismo de protección

Falta de flexibilidad y fiabilidad

El mecanismo de protección desarrollado está únicamente pensado para ser integrado en sistemas cliente/servidor. Por otro lado el núcleo de seguridad que ofrece el sistema situado en el núcleo del sistema operativo tan sólo sirve para validar la dirección del servidor pero la comprobación de permisos se hace en el servidor, y esta operación corre a cargo del usuario programador, que debe implementar esta comprobación. Si la implementación no se hace correctamente se puede producir un agujero de seguridad.

Sin previsión para movilidad

Los únicos objetos existentes en el sistema están gestionados por servidores y sólo ellos son capaces de utilizarlos, de manera que no está pensado para que los objetos se muevan de una máquina a otra. Por otra parte el mecanismo de protección está asociado al servidor y por tanto los objetos individuales no pueden trasladarse a otro sitio.

Falta de uniformidad y generalidad

No se trata de un mecanismo uniforme. El mecanismo de protección sólo sirve para acceso a objetos gestionados por servidores. No sirve para dar protección a objetos creados por el usuario que se comuniquen entre sí, a no ser que estén implementados detrás de un servidor

correspondiente que los gestione. El propio núcleo del sistema operativo ya no utiliza este mismo mecanismo de protección, sino que sigue los clásicos mecanismos de llamadas al sistema.

Granularidad gruesa

Los objetos a los que protege no son de cualquier granularidad, sino que se limitan a objetos de granularidad gruesa que son los que manejan los servidores.

Incumplimiento del principio de mínimo privilegio

El uso de capacidades permite que se cumpla el principio de mínimo privilegio, sin embargo, Amoeba no obliga a comprobar la protección en cualquier caso. El programador puede crear aplicaciones en las que se comuniquen objetos sin ningún tipo de protección.

Número fijo de operaciones protegidas

La protección de los objetos es al nivel de operaciones, lo cual supone mucha flexibilidad, sin embargo, el mecanismo sólo tiene posibilidad de proteger un máximo de 8 operaciones del objeto. Esto impide el uso del mecanismo para cualquier tipo de objeto que se desee, que en principio debería poder tener cualquier número de operaciones. Esto resulta demasiado restrictivo.

Sobrecarga por almacenamiento de capacidades en el espacio del usuario

Las capacidades se sitúan en el espacio de direcciones del usuario, siendo de este modo más fáciles de manejar. No obstante, esto supone una sobrecarga adicional, necesaria para asegurar que el usuario no falsifica o manipula las capacidades. En cada invocación a un método de un objeto, el servidor debe aplicar la función de verificación de que la capacidad no ha sido falsificada. Esta sobrecarga puede ser asumida siempre y cuando la granularidad de los objetos sea grande, y de hecho, Amoeba está pensado para gestionar servicios que manipulan objetos de granularidad grande, pero no funcionaría con sistemas más sofisticados y flexibles que gestionen objetos de granularidad fina.

Revocación parcial

Es posible una revocación de las capacidades, aunque esta revocación no puede ser selectiva. Modificando la clave que guarda el proceso servidor para un objeto dado se impide el posterior acceso a ningún proceso aunque éste contenga capacidad para ello. Sin embargo no es posible impedir el acceso a ciertos procesos determinados. O se impide el acceso a todos o no se permite a ninguno.

Re-implementación del mecanismo de protección en cada nuevo servidor

El mecanismo de protección va unido a cada proceso servidor. Este hecho es interesante porque significa que la protección no está centralizada sino que se mantiene distribuida, lo cual facilita la escalabilidad. Sin embargo, la creación de un nuevo servicio implica la implementación del mecanismo, dejándolo todo en manos del programador del servicio. Esto puede desvirtuar la filosofía del mecanismo.

Paradigma procedimental, no orientado a objetos

Por otra parte, el sistema no ha sido pensado para proporcionar soporte para objetos del usuario, sino que se apoya en el paradigma procedimental clásico, en el que el usuario dispone de procesos que mantienen su independencia a través de su espacio de direcciones y tan sólo se modifica la concepción clásica en la posibilidad de utilización de los distintos servicios que se le ofrezcan.

7.3 Sistema Operativo Choices

7.3.1 Características generales del sistema

El sistema Choices [CIM+93] de la Universidad de Illinois en Urbana-Champaign es un Sistema Operativo con un diseño interno orientado a objetos, e implementado en C++. El núcleo es un conjunto dinámico de objetos instanciados a partir de clases.

De cara al exterior, soporta aplicaciones paralelas y distribuidas, y ofrece interfaces de aplicación tanto orientadas a objetos como clásicas. Además proporciona soporte para objetos: definición dinámica de clases y herencia, y permite la interoperabilidad entre objetos de manera uniforme, independiente de la localización de éstos.

En su diseño, se crea una capa de adaptación entre el hardware y el software, orientada a objetos, para que el núcleo del sistema operativo actúe como si debajo tuviera un entorno hardware orientado a objetos.

El diseño de este sistema ha sido organizado mediante una jerarquía de clases, con clases abstractas, que especifican el comportamiento general, y clases concretas, que refinan la implementación y son usadas para especificar versiones particulares. De este modo, puede generarse una familia de sistemas operativos, con diferentes características.

7.3.2 Mecanismo de protección

La protección en Choices [CM90] está basada en una combinación de **objetos representantes** (*proxies*), **listas de control de acceso** y **servidores de nombres**.

El mecanismo de protección actúa cuando se intenta el acceso a una operación de un objeto, y está asociado al servidor de nombres, que se encarga de proporcionar el mecanismo de invocación de métodos. Aunque externamente el mecanismo de invocación es uniforme e independiente de la localización (local, o remota) de los objetos, internamente se establecen dos niveles diferentes de invocación entre objetos:

- **Situados en el mismo espacio de direcciones**. En este caso el sistema no media en la invocación del objeto, sino que ésta la realiza el propio lenguaje. No existe ningún mecanismo de protección que controle la invocación de unos objetos a otros dentro del mismo espacio de direcciones.
- **Situados en diferentes espacios de direcciones** dentro de la misma máquina. En este caso el servidor de nombres se ocupa de la localización del objeto, y también de la protección, mediante la consulta de la lista de control de acceso asociada a cada objeto, que indica si el cliente tiene acceso o no a dicho objeto. En caso afirmativo, el servidor de nombres crea un objeto *proxy*, con las operaciones del objeto servidor, que se mapea en el espacio de direcciones del cliente (bien estáticamente, bien dinámicamente bajo demanda). En tiempo de ejecución la llamada se efectúa sobre el objeto *proxy*, que se encarga de establecer la comunicación, mediante algún mecanismo (paso de mensajes, memoria compartida, etc.) con el objeto invocado.

  En el caso de que el objeto invocado se encuentre en una máquina diferente, el procedimiento es el mismo, aunque el *proxy* efectuará una RPC para establecer con él la comunicación.

El mecanismo de protección actúa al nivel de objetos y no de operaciones o métodos dentro de los objetos. La obtención del *proxy* permite el acceso a cualquier método del objeto de forma indiscriminada.

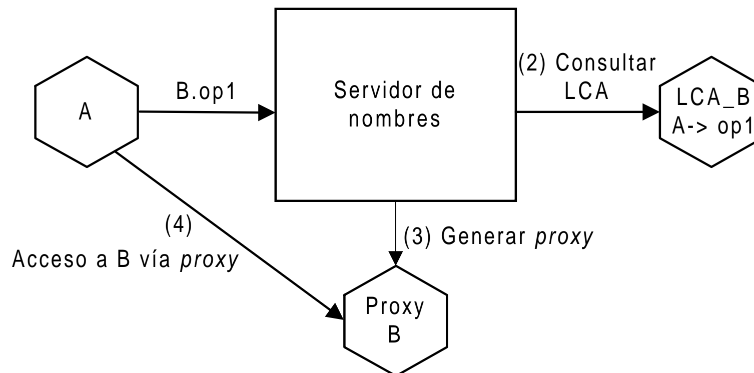


Figura 7.5: Funcionamiento del mecanismo de protección. Enlace del representante (proxy) con la aplicación.

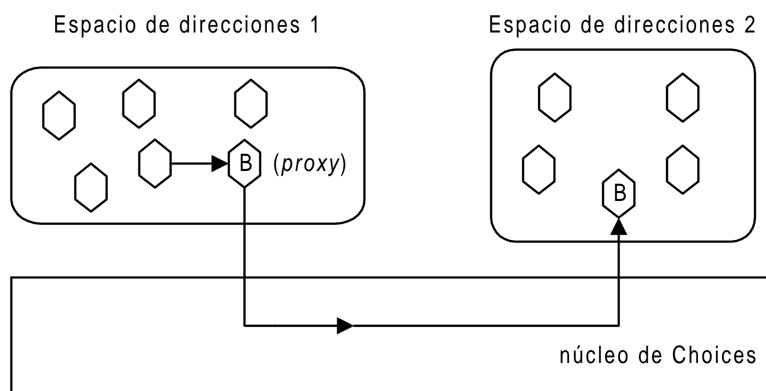


Figura 7.6: Acceso a un objeto a través de su representante (proxy)

7.3.3 Crítica del mecanismo de protección

Protección de grano grueso y falta del principio de mínimo privilegio

A pesar de que el mecanismo controla la seguridad de interacción entre objetos, este control se establece en el nivel grueso de objetos y no en el nivel fino de operaciones sobre los objetos. No es posible restringir el acceso a ciertas operaciones y permitir otras. Este hecho restringe considerablemente el poder de computación del sistema. En la mayoría de las ocasiones un control en el nivel de métodos resulta imprescindible.

Esto ya hace que sea imposible cumplir el principio del mínimo privilegio. Además, esta carencia se ve agravada, puesto que el sistema no protege los objetos cuando están dentro del mismo espacio de direcciones, permitiéndoles que interactúen libremente entre sí.

Falta de uniformidad

No es un mecanismo de protección uniforme, porque no protege los objetos situados en el mismo espacio de direcciones. En este sentido, el sistema continúa con la concepción tradicional de espacios de direcciones como modo de protección, y lo combina con el uso de listas de control de acceso cuando se pretenden hacer saltos entre espacios de direcciones.

Imposibilidad de revocación

Una vez que el servidor de nombres devuelve el *proxy* para permitir el acceso a un objeto, ya es posible el acceso al nuevo espacio de direcciones sin que tengan que efectuarse más

comprobaciones, puesto que ya se ha establecido el enlace. Esto significa que si se intenta eliminar el acceso ya no tendrá efecto, pues supone la eliminación de la entrada en la lista de control de acceso que posee el servidor de nombres, pero esta lista ya no se consultará hasta nuevos intentos de invocación por parte de objetos de otros programas.

7.4 Sistema Operativo Guide

7.4.1 Características generales del sistema Guide

Guide [Hag94], 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 otro ejemplo de 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 orientada 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 soporta un modelo de objetos pasivos, donde los objetos se ejecutan dentro de tareas, que disponen de un espacio de direcciones virtual. Los objetos de diferentes propietarios se asocian a contextos diferentes.

7.4.2 Mecanismo de protección

El mecanismo [HKR92] se basa en el uso de **listas de control de acceso** y en el **concepto de usuario**. El mecanismo se integra en el esquema de direccionamiento de los objetos y va unido al enlace dinámico que se realiza con la llamada a un método de un objeto. La protección se efectúa únicamente con la primera llamada al método.

7.4.2.1 Vistas

Se define la noción de **vista** como un conjunto de métodos autorizados correspondientes a una clase. Las vistas son por tanto una restricción de la interfaz de la clase, que se traduce en un conjunto de permisos sobre el objeto. Así mismo, se define una lista de control de acceso para cada objeto, que asocia una vista con cada usuario o grupo de usuarios. El dominio de protección va asociado por tanto al usuario o conjunto de usuarios.

Cada clase tiene asociada una tabla de vistas con una entrada por cada método de cada vista existente. Esta entrada tendrá un enlace al código del método si la vista permite su acceso, y estará vacía en caso contrario.

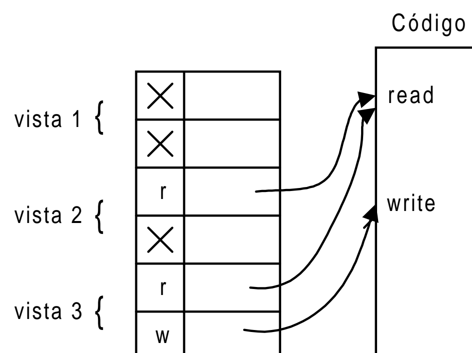


Figura 7.7: Tabla de vistas

7.4.2.2 Protección en el enlace dinámico

La protección se realiza en función del usuario. Cuando se intenta acceder a la operación de un objeto, como ocurre en la siguiente figura (1), en tiempo de ejecución se efectúa un enlace dinámico de la siguiente manera. Se busca en la lista de control de acceso del objeto la vista correspondiente a ese usuario (2). En función de esa vista, se accede a la tabla de enlace asociada a la clase (3), para el método invocado. A través de esta entrada se obtiene el enlace al método (4), si es que la vista lo permite, estableciéndose así el enlace dinámicamente que permite invocar los métodos del objeto.

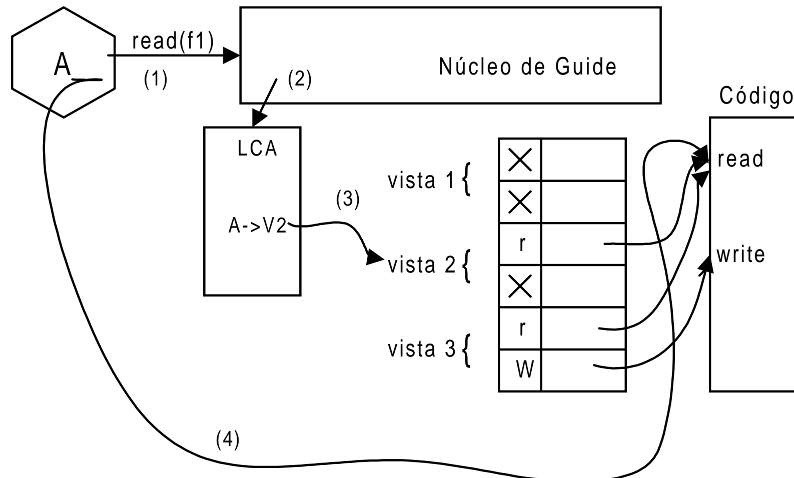


Figura 7.8: Mecanismo de protección en Guide

Este proceso sólo se realiza con la primera invocación al método. Sucesivas invocaciones utilizan el enlace previamente localizado (4), haciendo así que el mecanismo de protección sólo sea necesario para la primera invocación al método, y por tanto, no disminuya en exceso el rendimiento.

7.4.2.3 Resolución del problema de la delegación

Cuando la invocación de objetos involucra a diferentes propietarios ello implica un cruce de contexto. La resolución de este problema se realiza de forma análoga al bit *setuid* del Unix.

Cada objeto tiene una etiqueta formada por un único bit. Cuando se llama a un objeto, si este bit está a 1, se produce un cambio de contexto, y el nuevo usuario es el propietario de este objeto. En caso contrario, se ejecuta sobre el mismo contexto.

7.4.3 Crítica del mecanismo de protección

Falta de principio del mínimo privilegio y de mediación total

El mecanismo de protección sólo interviene cuando se accede a objetos situados dentro de contextos diferentes. Dentro del mismo contexto se permite el libre acceso. Esto último hace que la mediación no sea total, sólo se media en el acceso a objetos de otros contextos, y además sólo en la primera invocación.

Imposibilidad de revocación

Al realizarse la comprobación de protección sólo para la primera llamada, el resto de las llamadas a un método se realizan mediante un enlace a la dirección anteriormente recibida que no es posible revocar.

Solución del problema de la delegación parcialmente satisfactoria

La solución de usar un bit en cada objeto para este propósito es similar al concepto del bit *setuid* en Unix, y tiene todos los inconvenientes de esta última que se mostraron en el capítulo 3.

Falta de uniformidad y ahorro de mecanismos

Existe también una pequeña falta de uniformidad y ahorro de mecanismos puesto que para resolver el problema de la delegación se recurre a un mecanismo adicional.

7.5 Sistema Operativo Grasshopper

7.5.1 Características generales del sistema

Grasshopper [DBF+94] es un Sistema Operativo cuya característica principal es la de dar soporte a almacenamiento persistente, y que ha sido diseñado para funcionar en estaciones de trabajo convencionales. Este sistema ha sido desarrollado por investigadores de las Universidades de Adelaida y Sydney (Australia).

Para proporciona un entorno que soporte a la persistencia ortogonal completa, Grasshopper define tres abstracciones fundamentales: Contenedores, que son una representación abstracta de almacenamiento (zonas de datos), *loci* que representan acciones dentro del sistema (flujo de ejecución) y capacidades, utilizadas para representar los permisos de acceso a un objeto.

Los contenedores son las entidades persistentes, que reemplazan a los clásicos conceptos de espacio de direcciones y sistema de ficheros. Contenedores y *loci* son conceptos ortogonales. El sistema consiste en un conjunto de contenedores que pueden tener *loci* ejecutándose sobre ellos.

7.5.2 Mecanismo de protección

7.5.2.1 Capacidades segregadas para proteger contenedores y *loci*

Grasshopper utiliza capacidades segregadas como mecanismo de protección. Se trata más concretamente de un modelo mixto de capacidades en el que la lista de permisos es sustituida por una clave como se explica a continuación.

Los elementos a proteger son los contenedores y los *loci*. Por lo tanto, la protección con capacidades se usa exclusivamente para proteger los segmentos de memoria.

7.5.2.2 Información que poseen las entidades a proteger

Cada entidad a proteger (contenedor o *locus*) posee los siguientes elementos:

- **Tabla de capacidades** que contiene las capacidades poseídas por esa entidad. Cada entrada en la tabla contiene una clave y un puntero a la entrada en la tabla de grupos de permisos del objeto sobre el que se posee la capacidad.
- **Tabla de grupos de permisos**, que llevan el control sobre el acceso a esa entidad (qué otras entidades pueden acceder a ella). Se podría considerar como una lista de control de acceso. Cada entrada contiene un conjunto de permisos, entre los que cuenta con permisos específicos para la capacidad, como destruirla, copiarla, etc. Estas dos tablas son gestionadas por el sistema operativo.
- **Capref (referencia de capacidad)**. Es la forma en que aparecen las capacidades al programador de aplicaciones. No están protegidas por el sistema. Contienen una marca (*flag*) y una clave. La clave se refiere a la entrada en la tabla de capacidades.

7.5.2.3 Modo de funcionamiento: mixto capacidades / listas de control de acceso

Dentro de un programa se hace referencia a un objeto a través de la *capref*, la cual contiene una entrada en la tabla de capacidades. A través de esa entrada el sistema accede a una entrada en la tabla de grupos de permisos asociada con el objeto invocado. La entrada en la tabla de grupos de permisos proporciona los permisos que posee el cliente, y si entre ellos está la operación invocada, el sistema efectúa la invocación, en caso contrario la deniega.

Se trata de un mecanismo mixto entre capacidades y listas de control de acceso, en el que la capacidad en lugar de tener directamente los permisos, tiene una clave de acceso a otra tabla donde se guardan los permisos, dependiendo esta última tabla del objeto servidor.

La protección la realiza el sistema operativo y las capacidades están segregadas, por tanto no son accesibles directamente por el usuario, lo que evita la falsificación y por tanto la necesidad de sistemas especiales para comprobar que no se falsifican.

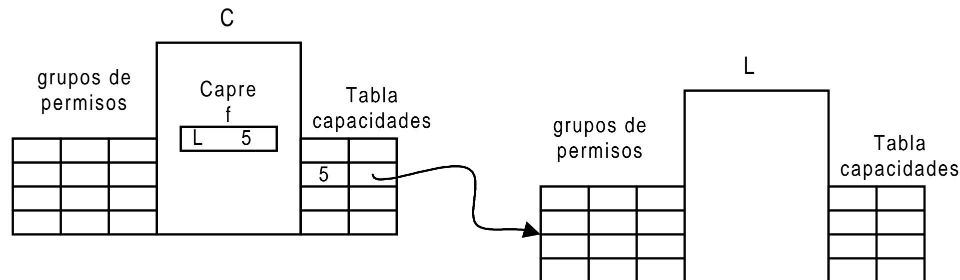


Figura 7.9: Control de acceso en Grasshopper

7.5.3 Crítica del mecanismo de protección

Revocación selectiva parcial

El modelo mixto permite la revocación selectiva para todos aquellos que tengan la misma entrada en la tabla de grupos de permisos. Es por tanto una revocacion selectiva parcial, no puede hacerse para un cliente individual.

Evitación de la falsificación a costa de flexibilidad

Se evita la comprobación de falsificación pues las capacidades no son directamente accesibles por el usuario y son gestionadas por el sistema en un lugar protegido. Sin embargo esto hace que la gestión de capacidades por el usuario sea más compleja, perdiéndose flexibilidad para que el usuario las intercambie libremente, sobre todo en entornos distribuidos.

Sobrecarga de las listas de control de acceso

Tiene los inconvenientes de sobrecarga de las listas de control de acceso, que necesitan ser consultadas en cada acceso. Por otra parte, las listas pueden volverse muy grandes si el número de entidades se incrementa, como suele suceder en un sistema distribuido.

Sin soporte para objetos

No se trata de un sistema que de soporte a objetos. Las abstracciones básicas que protege son de muy bajo nivel y muy lejanas a la semántica del modelo de objetos. El sistema de protección no es homogéneo sino que tan sólo se emplea dentro del sistema operativo para la gestión de la persistencia.

7.6 Sistema Operativo Hydra

7.6.1 Características Generales

El sistema operativo Hydra [WLH81] es un sistema experimental, diseñado específicamente en la Universidad de Carnegie Mellon, para investigar la plataforma hardware denominada c.mmp. Se trata de un sistema operativo distribuido, cuyo objetivo fundamental de diseño es la separación entre política y mecanismo. Es decir, separar los mecanismos que son implementados en el núcleo del sistema operativo, de las políticas, que se implementan como programas de usuario. Ejemplos de esta separación son la planificación y la paginación. Esta separación es especialmente importante en el esquema de protección, el cual está basado en capacidades.

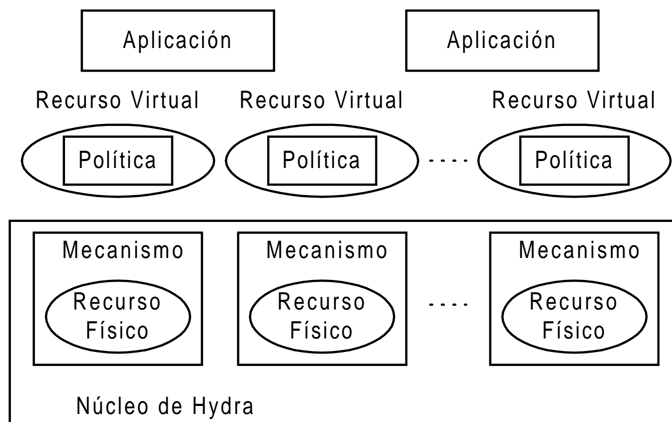


Figura 7.10: Arquitectura del Sistema operativo Hydra

Otra característica de diseño del Hydra es que el núcleo soporta tipos abstractos de datos (objetos). Cada recurso gestionado es un objeto denominado recurso virtual.

7.6.2 Mecanismo de protección

El mecanismo básico de protección que utiliza Hydra está basado en **capacidades** desarrolladas en un hardware específico en la universidad de Carnegie Mellon. Está implementado dentro del núcleo, a diferencia de las políticas que deberán implementarse externamente.

Ciertos registros de la máquina permiten el uso de capacidades a través de un conjunto restringido de operaciones (cargar, almacenar, copiar, restringir permisos y llamar).

Una capacidad está compuesta por un identificador único y un campo que describe los métodos autorizados entre todos los declarados en el tipo del objeto referenciado. El estado de un objeto está compuesto de datos y de una lista de capacidades.

La unidad de protección es el objeto, que representa un recurso. Tanto los objetos del sistema como objetos creados por el usuario podrán utilizar este mecanismo de protección.

Las capacidades son manipuladas únicamente por el núcleo. Así por ejemplo un "espacio local de nombres" es un objeto que representa un programa en ejecución, que cuenta con un conjunto de capacidades (dominio de protección).

El sistema operativo define tres tipos de objetos (procedimientos, semáforos y procesos), y además está preparado para manejar tipos de objetos arbitrarios del espacio del usuario.

Cada objeto tiene un subconjunto de operaciones que manipulan la parte de datos del objeto (*get/put/add*), y otro conjunto (*load/store/append/delete/copy/create*) que manipulan la lista de

capacidades del objeto. Las operaciones *store* y *append* utilizan una máscara para controlar la propagación de permisos (comparando con el indicador (*flag*) *copy* de la matriz de acceso, que controla si la capacidad se puede copiar o no). Estas operaciones son las herramientas fundamentales utilizadas para especificar la política.

El núcleo soporta dos tipos de permisos: genéricos y auxiliares. Los permisos genéricos pueden ser aplicados a cualquier objeto, mientras que los auxiliares son definidos por tipos de objetos particulares. El núcleo emplea una máscara de 24 bits para identificar los permisos. Los permisos genéricos tienen posiciones fijas en la máscara, mientras que los auxiliares están asignados de acuerdo a los objetos y a las políticas para manejarlos.

Los objetos se definen implementando un subsistema compuesto de un conjunto de procedimientos y una lista de capacidades. El subsistema define un nuevo tipo para el objeto. Los subsistemas son el mecanismo que permite definir una política como un objeto, donde el objeto puede ser invocado desde otros objetos, tiene un conjunto específico de permisos, etc. El núcleo maneja la creación del objeto, el tipo y la comprobación de permisos, y la amplificación cuando un objeto es invocado; las acciones particulares especificadas por el objeto se determinan por la definición del subsistema.

Un proceso que se ejecuta en un objeto sólo puede usar las capacidades almacenadas en la lista de capacidades del objeto y algunas capacidades recibidas como parámetros. Como cada llamada a un método cambia el espacio de capacidades (o espacio de direcciones) del proceso actual, cada objeto es gestionado en un dominio de protección diferente.

Una característica del mecanismo de protección es que realiza un control sobre la propagación de las capacidades, limitando ésta a través de la comprobación de un bit de copia que aparece entre los permisos de éstas.

El mecanismo base está diseñado para permitir soluciones a nivel de política a la sospecha mutua, confinamiento y revocación dinámica de permisos.

7.6.2.1 Parámetros plantilla de los procedimientos

Los procedimientos contienen parámetros plantilla (*templates*) que indican el tipo de capacidades que deben ser pasadas a éste. Cuando se invoca al procedimiento el núcleo comprueba que en la invocación se le pasen las capacidades con los permisos adecuados, en cuyo caso construye una capacidad para el procedimiento conteniendo los permisos especificados en el patrón, que pueden ser más que los especificados en la invocación. El llamado puede disponer de más permisos que el llamador, sin embargo no ocurre al revés. Esto es un factor clave para ofrecer flexibilidad en Hydra y para cumplir el principio del mínimo privilegio.

7.6.3 Crítica del mecanismo de protección

El sistema Hydra es uno de los pioneros en la utilización de capacidades y presenta muchos aciertos de diseño: separación entre política y mecanismo, dominios individuales para cada objeto con principio de mínimo privilegio, etc. Sin embargo algunos aspectos no son adecuados.

Utilización de hardware específico

La obligación de utilizar un hardware específico para gestionar las capacidades restringe la difusión del sistema a plataformas hardware heterogéneas.

Poca flexibilidad de las capacidades segregadas

La gestión de capacidades mediante operaciones especiales del sistema impide la falsificación de las mismas, aunque complica el trabajo en el espacio de usuario al tenerse que preocupar de un nuevo conjunto diferente de operaciones para las capacidades, diferente de las operaciones normales que realiza el usuario con los objetos.

Limitación del número de permisos

Aunque el número de operaciones que permite proteger dentro de los objetos es amplio, este número es fijo. Puesto que un objeto en principio puede tener tantas operaciones como necesite su semántica, esto restringe el soporte al modelo de objetos.

Complejo dentro de la sencillez de las capacidades

A pesar de la sencillez del modelo de capacidades, existe una complicación adicional del modelo. Se añaden bits de protección y construcciones especiales para solucionar problemas específicos, aunque en un entorno de aplicación del sistema determinado no se necesitase resolver estos problemas. Estos entornos sufren entonces los problemas de sobrecarga en el espacio y en el tiempo que implica esta funcionalidad no utilizada.

Sistema basado en objetos y no orientado a objetos

El sistema está basado en objetos simplemente, y no tiene toda la semántica normalmente asociada a un modelo orientado a objetos (herencia, etc.).

7.7 Sistema Operativo KeyKOS

7.7.1 Características generales del sistema

Se trata de un sistema implementado para soporte del servicio de la British Telecom creado en 1975 que estuvo en producción hasta 1983. Su arquitectura [Hard85] se basa en un micro-núcleo, desarrollado para soportar un entorno seguro, con compartición de datos. Ofrece diferentes interfaces del propio sistema KeyKOS o de emulaciones de otros sistemas operativos. Los datos del sistema se almacenan en memoria virtual persistente.

7.7.2 Mecanismo de protección

La parte del sistema que se encarga de la protección se denomina KeySAFE [RHB+86]. Ha sido diseñado para soportar políticas de seguridad discrecional y obligatorio, con un nivel de protección B3, según el libro naranja [DoD83]. El mecanismo pretende dar a la plataforma KeyKOS flexibilidad para soportar diferentes políticas de seguridad.

Para el control de acceso a los objetos del sistema utiliza capacidades. El sistema crea las capacidades y las aplicaciones pueden realizar duplicaciones de éstas.

7.7.2.1 Estructura de las capacidades

La capacidad tiene un campo de 8 bits usado para identificar el tipo de protección sobre el objeto (clase de capacidad). Cuando un proceso envía una capacidad puede colocar este campo a un determinado valor, permitiendo dividir a los clientes en clases de servicio o niveles privilegiados.

Las capacidades son segregadas, y sólo el micronúcleo tiene acceso a ellas.

7.7.2.2 Monitor de referencia

La idea clave, sobre la que se organiza la protección, es el concepto de compartimento. Se define un compartimento para cada sujeto. Todas las capacidades de un compartimento serán bien para objetos del mismo compartimento, en cuyo caso se aplica la protección habitual de las capacidades; o para objetos de otros compartimentos, en cuyo caso su uso pasa por un monitor de referencia, que aplica las reglas de seguridad necesarias.

El sistema proporciona un mecanismo de indirección, que permite que el acceso sea rescindido por el monitor de referencia cuando lo considere oportuno. Esto también permite al monitor de referencia forzar el acceso a sólo lectura, y prohibir el paso o recepción de otras capacidades.

Un sujeto no puede conseguir acceso a un objeto fuera de su compartimento sin pasar por el monitor de referencia.

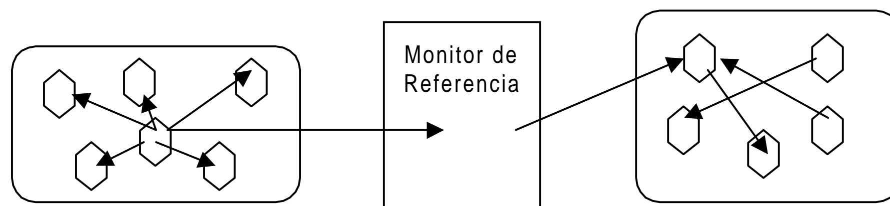


Figura 7.11: Monitor de referencia

Inicialmente, la única capacidad dirigida hacia un objeto fuera del compartimento es la capacidad guarda, que permite el acceso al propio monitor de referencia. Esta capacidad permite al sujeto en un compartimento crear objetos con etiquetas de seguridad, especificar listas de control de acceso, y hacerlas disponibles a otros compartimentos a través del monitor de referencia (sujeto a control de acceso discrecional u obligatorio). La guarda es también el medio por el cual los sujetos pueden obtener capacidades a objetos creados en otros compartimentos. Cada compartimento tiene su propia guarda, que permite al monitor de referencia identificar al sujeto.

Las capacidades nunca son transferidas directamente de un compartimento a otro. En lugar de ello, el monitor de referencia crea una capacidad intermedia.

Así pues, existe un mecanismo básico de protección, las capacidades, que se complementa con un monitor de referencia que implementa la políticas de seguridad. Entre los aspectos que gestiona el monitor de referencia están la posibilidad de revocación de capacidades, mediante el uso de indirección, y el control de la propagación de capacidades. En un entorno de red, el monitor de referencia puede implementar un rango de políticas para tratar con peticiones de la red.

7.7.3 Crítica del mecanismo de protección

Falta de flexibilidad de las capacidades segregadas

Como ya se mencionó anteriormente, las capacidades segregadas introducen una cierta inflexibilidad al no permitirse su manipulación normal dentro de las estructuras normales de usuario.

Sin soporte para el modelo orientado a objetos

No hay soporte para toda la semántica del modelo de objetos. El sistema simplemente tiene el concepto de puntos de entrada en un espacio de direcciones, que se podría asimilar a un objeto con métodos, pero otros conceptos como la herencia no existen.

Poca flexibilidad adicional del monitor de referencia

La posibilidad de ampliar las capacidades implementando mecanismos de indirección y revocación selectiva mediante un monitor de referencia que se interpone entre el cliente y el servidor es muy interesante. Sin embargo, no es sencillo cambiar la funcionalidad del monitor de referencia dentro del sistema, ni esta puede extenderse fácilmente por parte del usuario.

Por otro lado, el sistema obliga siempre a la existencia del monitor de referencia, aunque en una aplicación concreta no se necesite su funcionalidad. Esto hace que haya que pagar por ésta aunque no se necesite.

7.8 Sistema Operativo Mungi

7.8.1 Características generales del sistema

En la Universidad de New South Wales se diseña el Sistema Operativo Mungi [VRH93]. Se trata de un sistema de gestión de memoria virtual global persistente. Proporciona un único espacio de direcciones, el cual contiene la memoria virtual de todos los procesos en todos los nodos de un sistema de computación distribuida.

Los objetos son segmentos de memoria virtual, agrupados en páginas, las cuales son las unidades básicas de protección. La distribución es transparente y los datos y los procesos pueden migrar de unos nodos a otros.

7.8.2 Mecanismo de protección

En un espacio de direcciones único, todos los objetos son visibles por cualquier proceso, sin que sea necesaria una interacción explícita del sistema. Por tanto es necesario un mecanismo de protección de los objetos para que sólo se permitan ciertos accesos.

7.8.2.1 Capacidades *password*

Mungi utiliza un mecanismo de protección basado en capacidades *password*.

Protección de operaciones elementales

La protección se realiza sobre los objetos del sistema, es decir, sobre segmentos de memoria virtual formados por páginas contiguas. Las operaciones de estos objetos sobre las que se va a ejercer la protección son cuatro: lectura (r), escritura (w), ejecución (x) y destrucción (d).

Tabla de objetos global

Cuando se crea un objeto, el sistema guarda en una estructura global (tabla de objetos) información sobre su dirección de comienzo, su longitud y una clave que se le asigna. Además devuelve al propietario del objeto una capacidad, formada por la dirección y la *password* con todos los permisos sobre el objeto.

Cuando se desea acceder al objeto deberá presentarse la capacidad, y el sistema comprueba si efectivamente existen permisos para la operación deseada.

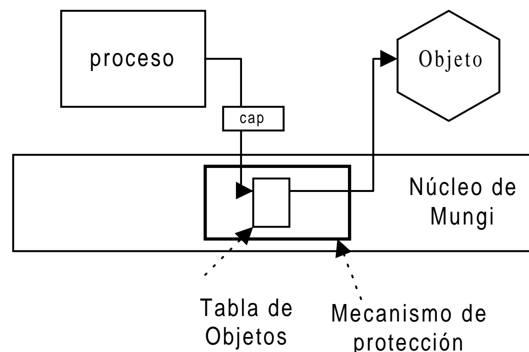


Figura 7.12: Mecanismo de protección del sistema Mungi

Capacidades restringidas

Cuando se crea una capacidad, el sistema deriva un conjunto de cinco capacidades restringidas cada una con un número de permisos menor, mediante una función de único sentido, de forma análoga a como lo hace el sistema Amoeba. Estas capacidades se guardan en la tabla de objetos.

El propietario de una capacidad puede pedir al sistema que cree otra capacidad con otra *password*, y todas sus capacidades derivadas. También puede pedir al sistema que elimine una capacidad de la tabla de objetos, realizando así una revocación selectiva.

7.8.2.2 Sistema jerárquico de seguridad

Mungi soporta un sistema de seguridad jerárquico, similar a Unix, donde existen permisos para el propietario, el grupo y el resto. Para ello las capacidades se almacenan en un árbol de capacidades, que es un único objeto compartido por todos los núcleos en el sistema. Cada nodo del árbol se denomina nodo de protección y está enlazado con una lista de capacidades, que está gestionada por el usuario.

Las capacidades para los objetos públicos (accesibles por cualquiera) se almacenan en un nodo raíz del árbol, mientras que las capacidades privadas de cada usuario se almacenan en un nodo hoja. Las capacidades de los grupos se almacenan en nodos intermedios del árbol.

Dominios de protección

Cuando el sistema busca una capacidad, lo hace partiendo del nodo hoja del usuario hacia arriba hasta la raíz. El conjunto de capacidades encontradas al atravesar el camino se denominan "dominio de protección normal".

Cuando un usuario entra en el sistema, se crea el dominio de protección activo, que es el conjunto de listas de capacidades encontradas por el sistema en el dominio de protección regular, y se asocia con su bloque de control de proceso (PCB). A partir de él, el usuario puede añadir o eliminar capacidades como le parezca.

Además de la lista de capacidades, cada nodo puede tener un manejador de fallo de protección que proporcione una estrategia de búsqueda alternativa.

7.8.3 Crítica del mecanismo de protección

Objetos con poca semántica

Los objetos no son objetos creados mediante un lenguaje de programación orientada a objetos, sino simples espacios de memoria sin apenas semántica. En realidad no se da soporte a objetos, si no a abstracciones mucho más elementales.

Protección ligada al concepto de usuario

La protección está íntimamente ligada al concepto de usuario, lo que hace que deba utilizarse esta abstracción de usuario incluso en aplicaciones en la que no es necesaria.

Agrupamiento de permisos poco flexible: sin mínimo privilegio

Los permisos se agrupan de forma tradicional en permisos para el propietario, grupo y resto de los usuarios. Aunque se gana así en eficiencia se pierde flexibilidad puesto que hay combinaciones que son muy difíciles de expresar con esos tres niveles de agrupamiento. Esto hace que no se pueda cumplir el principio de mínimo privilegio.

Número fijo de operaciones protegidas y de bajo nivel

Sólo se protegen las operaciones básicas con las abstracciones elementales del sistema: lectura, escritura, ejecución y eliminación. Esto es de un nivel demasiado bajo y poco flexible para soportar un sistema de objetos con semántica completa, en la que los objetos pueden tener un número arbitrario de operaciones de alto nivel.

7.9 Sistema Operativo Opal

7.9.1 Características generales del sistema

Opal [CLF+93] es un sistema operativo que proporciona un espacio único de direcciones, en el cual los objetos están únicamente identificados por sus direcciones, que está siendo desarrollado en la Universidad de Washington, Seattle. Actualmente está implementado como un prototipo basado en el micronúcleo Mach 3.0.

7.9.2 Mecanismo de protección

El concepto de objeto para Opal es el de zona de memoria. La unidad básica de almacenamiento y protección es el segmento. Los segmentos de memoria pueden ser persistentes. El soporte de espacios de direcciones únicos tiene la ventaja de facilitar la compartición de estructuras de datos entre procesos.

Se utilizan las capacidades *password* para proteger los objetos.

7.9.2.1 Dominios de protección

El control de acceso a los objetos está gestionado a través de un dominio de protección que permite a un hilo tener acceso a un conjunto específico de páginas en un instante dado. Un dominio de protección es un conjunto de permisos a segmentos en un instante dado. Un hilo se ejecuta en un dominio de protección y tiene acceso a los permisos conferidos a dicho dominio.

Las capacidades *password* se utilizan para controlar el acceso a recursos tales como dominios de protección y segmentos.

7.9.2.2 Portales

Si se produce una llamada a un segmento en otro dominio se utiliza para ello los portales. Un portal consiste en un punto de entrada a un dominio. Cualquier hilo conocedor de un identificador de portal puede transferir el control al dominio asociado al portal.

Equivalencia con Amoeba

La capacidad, por tanto, consiste en un identificador de portal de 64 bits, una dirección de objeto de 64 bits y un campo de validación, aleatorio, de 128 bits. Este método es exactamente el mismo que el empleado por Amoeba.

El identificador del portal se usa para hacer una llamada al servidor que maneja el segmento o dominio representado por la capacidad. El campo de validación realiza las funciones de identificar el conjunto de permisos conferidos por la capacidad, evitando falsificación y permitiendo revocación. El servidor garantiza o deniega el acceso al recurso representado por la capacidad.

7.9.3 Crítica del mecanismo de protección

Desde el punto de vista de la protección, puede considerarse a Opal como una mezcla entre el sistema Amoeba (protección por capacidades *password*) y Mungi (los objetos son simples zonas de memoria). Por tanto los inconvenientes son similares:

- Sobrecarga causada por el almacenamiento de capacidades con contraseña en el espacio del usuario
- Número fijo de operaciones protegidas y de bajo nivel
- Objetos con poca semántica

7.10 Sistema Operativo Spin

7.10.1 Características Generales

El sistema operativo SPIN [BSP+95] tiene como objetivo desarrollar un sistema en el que el núcleo del sistema operativo pueda extenderse con seguridad mediante código de usuario (en este caso mediante el lenguaje "seguro" Modula-3). Es decir, permitir que el usuario pueda añadir funcionalidad al sistema operativo introduciendo código en el espacio del sistema. Al existir una clara separación entre el espacio del usuario y el del sistema, esto además permite un aumento del rendimiento al no tener que realizarse cambios de contexto entre los espacios.

Sin embargo, la adición de código de usuario al sistema dentro del espacio del núcleo presenta un problema de seguridad. Este código de usuario podría bien por errores de programación, o malintencionadamente, corromper estructuras y/o código del sistema operativo.

Para protegerse frente a las extensiones, SPIN divide los servicios en dominios y hacen que las extensiones sólo puedan acceder dentro del dominio en el que se incluyen. Sin embargo, no hay control de grano fino dentro del dominio, la extensión puede acceder a todos los servicios del dominio, a no ser que cada servicio implemente su propia política de acceso cuando son extendidos.

El sistema utiliza capacidades (más restringidas que las convencionales) como mecanismo de protección entre dominios. Las capacidades se implementan usando punteros a nivel del lenguaje "seguro" en el que se programan las extensiones, de tal manera que sólo teniendo acceso al puntero dentro del lenguaje se puede acceder y únicamente a las operaciones que permita el tipo del puntero (aunque no se puede seleccionar un subconjunto de ellas).

7.10.2 Crítica del mecanismo de protección

El sistema SPIN es interesante en tanto que es un ejemplo de un sistema que puede ser extendido por código de usuario, al igual que se pretende que lo sea el sistema integral. De todas maneras, el sistema de extensión no parece tan flexible como necesita el sistema integral y en cuanto a la protección, también se presentan deficiencias.

Falta de uniformidad

Este mecanismo de protección se utiliza únicamente para proteger al sistema frente a sus extensiones por el usuario. No existe un mecanismo de protección genérico que se utilice para la protección de código normal de usuario.

Falta de principio de mínimo privilegio

Para limitar los efectos de las extensiones, se agrupan los servicios que pueden ser extendidos en dominios. Sin embargo, dentro de un dominio, una extensión potencialmente puede acceder a su antojo a todos los recursos del mismo, los necesite o no.

Para controlar la interacción entre dominios utiliza un mecanismo adicional, utilizando punteros del lenguaje seguro, de manera que no se pueden utilizar otras operaciones que las definidas por el tipo. Sin embargo no se puede restringir aún más este conjunto de operaciones para controlar el acceso a operaciones individuales, apareciendo otro punto en el que se rompe el principio de mínimo privilegio.

No orientado a objetos

El sistema SPIN no está diseñado con el objetivo de la orientación a objetos, por tanto no da soporte directo a ningún elemento relacionado con la OO, y sus abstracciones son convencionales. Existe una separación clara entre el espacio del usuario y el del sistema.

7.11 Tabla resumen de las principales características de los sistemas operativos analizados

Nombre	Tipo de Protección	Elementos a proteger	Mecanismo
Amadeus	Listas de Control de Acceso al nivel de ámbitos. Usuario como dominio de protección.	Objetos definidos por el usuario	Al nivel del núcleo. Agrupación de objetos en ámbitos (conjunto de objetos fiables). Uso mecanismo en cambios de ámbito Control al nivel de operaciones del objeto
Amoeba	Capacidades *Password*.	Objetos definidos y gestionados por servidores. Ej: Ficheros, etc.	Servidor genera y comprueba capacidades en cada invocación. Tabla de objetos. Núcleo usa capacidades para buscar servidor. Permisos hasta 8 ops. objeto.
Choices	Listas de Control de Acceso Usuario como dominio de protección	Objetos definidos por el usuario	Control sobre objetos de espacios de direcciones diferentes. Generación de representantes
Guide	Listas de Control de Acceso. Tablas de vistas: métodos autorizados Usuario como dominio de protección	Objetos definidos por el usuario	A nivel de núcleo Control en la primera invocación al objeto. Control de acceso entre objetos de diferentes espacios de direcciones
Grasshopper	Capacidades segregadas mixtas con LCA	Objeto: segmentos de memoria (contenedores y loci)	Capacidad vista por el usuario: (nombre-clave). Clave manejada por el núcleo. A través de ella, acceso permisos
Hydra	Capacidades segregadas, con hardware especial	Objetos del sistema y del usuario	A nivel de núcleo Control de operaciones del objeto y de la capacidad (propagación)
KeyKOS	Capacidades segregadas	Objetos del sistema y del usuario Del sistema: dispositivos, páginas, etc.	A nivel del núcleo Junto con el mecanismo de envío de mensajes. Monitor de referencia para establecer política por encima.
Mungi	Capacidades *Password*	Objeto: Segmento de memoria Compartición de objetos entre procesos	A nivel del núcleo. Tabla de objetos para comprobar la capacidad. Control de operaciones: rwx
Opal	Capacidades *Password*	Objeto: Segmento de memoria Espacio de direcciones único	Los servidores manejan los objetos al estilo de Amoeba
Spin	Separación en dominios y uso de punteros con tipo	Recursos del espacio del sistema	Utilización de un lenguaje seguro

7.12 Conclusiones sobre los sistemas estudiados

Si bien estos ejemplos que aquí se describen son una muestra de algunos sistemas existentes, nos proporcionan una idea general de la situación de seguridad en el momento actual.

Los sistemas como Amadeus, Guide o Choices, pretenden introducir en el software de sistemas los conceptos de orientación a objetos, y dar soporte a la interoperabilidad de objetos definidos por el usuario, situados en espacios de direcciones diferentes e incluso dentro de un sistema distribuido. El problema de la protección es resuelto mediante el uso de listas de control de acceso y la protección es de grano grueso, aplicada solamente cuando se atraviesan las barreras proporcionadas por el clásico espacio de direcciones.

Los sistemas como Amoeba, Hydra y KeyKOS, pretenden también dar soporte a objetos del usuario, aunque no tan ligado a los puros lenguajes de programación. Se centran más en el objetivo de conseguir la gestión de un sistema distribuido. Utilizan capacidades para implementar la protección, que están ligadas a objetos de granularidad gruesa que no están estructurados mediante ningún modelo de objetos típico.

Los sistemas como Grasshopper, Mungi y Opal son sistemas especializados en la gestión de un espacio de memoria global persistente. Con este objetivo, utilizan un mecanismo de protección basado en capacidades (o capacidades mixtas en el caso de Grasshopper) para la protección de lo que ellos definen como objetos, que no son más que segmentos de memoria.

En general, los sistemas estudiados carecen de un mecanismo uniforme que proteja por igual cualquier acceso, y utilizan soluciones parciales, ajustadas en concreto al problema a resolver, y carentes de flexibilidad. Por otra parte, adolecen del principio de mínimo privilegio, pues la protección no se produce a nivel de granularidad fina.

Éstas y otras razones, que posteriormente serán esbozadas, han sido la motivación de este trabajo: la búsqueda de un sistema de protección completo, más flexible y adecuado a los requisitos de un sistema integral orientado a objetos.

8 Seguridad en la Plataforma Java

8.1 Evolución de Internet

El continuo crecimiento y popularidad de Internet ha conducido a una actividad frenética por parte de desarrolladores del Word Wide Web, pasando de estar compuesta por un conjunto de documentos estáticos HTML, con información de consulta desde cualquier parte del mundo a transformase en un complejo sistema distribuido.

Con el uso de la red de redes se ha popularizado la ejecución de código móvil[1], procedente de cualquier parte de la red y con ello la cooperación de usuarios a través de objetos compartidos. Actualmente es posible utilizar una variedad de sistemas de código móvil como Java [GJS96], JavaScript [FLA97], ActiveX [MC96], y Shockwave [Sch97].

8.2 Plataforma Java

La plataforma Java ha contribuido en gran medida al desarrollo de Internet. Con ella apareció la posibilidad de ejecutar en una máquina propia aplicaciones procedentes de servidores repartidos por la red. Esto supone la apertura de la computación hacia un modelo heterogéneo y distribuido.

La base de la plataforma es el lenguaje Java, el cual permite el desarrollo de aplicaciones distribuidas a través de un lenguaje orientado a objetos del tipo del C++. La principal innovación de Java es que permite una gestión más fácil de código móvil. El código generado por el compilador de Java es interpretado por la máquina virtual de Java, posibilitando la transferencia de código entre lugares heterogéneos. En particular, se han desarrollado varios navegadores (*browsers*) Web, como HotJava o Netscape, que encapsulan una máquina virtual de Java y son utilizados para cargar pequeñas aplicaciones en Java denominadas *applets* que están enlazadas en páginas HTML [DFW96].

8.3 El problema de la protección

Uno de los principales inconvenientes de esta apertura, y en general de cualquier tipo de computación distribuida es la protección. La posibilidad de que código ajeno, procedente de una máquina remota, pueda llegar a la máquina propia y ejecutarse en ella, implica la posibilidad de acceso por parte de dicho código a los recursos que la máquina posee, como por ejemplo, el sistema de ficheros, y por tanto, los datos. Es por ello, absolutamente imprescindible poner cota a este hecho y restringir la utilización de los recursos locales para que código remoto desconocido no pueda tener acceso a información privada o pueda provocar destrozos en la máquina local (caballos de Troya).

[1] Móvil en el sentido de que se descarga del servidor para ejecutarse en el cliente.

8.4 Tratamiento de la seguridad en Java

La seguridad de Java viene dada por una serie de mecanismos que incluyen a la seguridad de tipos del lenguaje, la verificación del *bytecode*, la comprobación de tipos en tiempo de ejecución, la separación del espacio de nombres mediante la carga y el control de acceso a las clases vía un gestor de seguridad.

Analizando los elementos que intervienen en la seguridad en Java, se puede agrupar en cuatro niveles la defensa contra posibles ataques en la plataforma Java: seguridad en el diseño del lenguaje, verificador de *bytecodes*, cargador de clases y protección del sistema de ficheros y del acceso a la red.

8.4.1 Seguridad en el diseño del lenguaje

El primer nivel de defensa es el propio lenguaje. El lenguaje Java fue diseñado pensando en evitar agujeros de seguridad que no contemplaban otros lenguajes como el C++.

El lenguaje no permite acceso directo al espacio de direcciones del programa: los objetos no son manipulados a través de punteros, sino de referencias a nivel del lenguaje. En Java todos los accesos a memoria deben ser hechos usando referencias a objetos.

El lenguaje es fuertemente tipado, implementando comprobaciones de tipo muy rigurosas en tiempo de compilación y ejecución.

Además de éstos, aspectos como la comprobación de límites de índices de arrays y cadenas, la eliminación del ahormado automático, o la recolección automática de basura, contribuyen a proporcionar un lenguaje más seguro.

8.4.2 Verificador de bytecodes

El segundo nivel de defensa es parte del entorno en tiempo de ejecución, el cual comprueba que el código a ejecutar (las instrucciones de la máquina virtual o *bytecodes*) sigue las reglas base de seguridad. Comprueba que no se falsifiquen punteros, no se violen restricciones de acceso, que las llamadas a métodos tengan los argumentos apropiados, verifica la conformidad de métodos entre el llamador y el llamado en el momento de la invocación, etc.

8.4.3 Cargador de clases

El tercer nivel de defensa forma parte también del entorno de ejecución. Los cargadores de clases se emplean para cargar clases procedentes de la red y contienen una serie de elementos que contribuyen a incrementar la seguridad. Así por ejemplo, son ellos los que se encargan de crear espacios de nombres diferentes para clases procedentes de localizaciones remotas diferentes, de forma que no se produzcan ambigüedades con posibles nombres de clases iguales aunque procedentes de lugares diferentes.

8.4.4 Protección del sistema de ficheros y del acceso a la red

El último nivel de defensa constituye la parte más compleja del problema de la seguridad en un sistema distribuido como el de Internet. Se trata de controlar el acceso al sistema de ficheros local, limitando éste en caso de aplicaciones remotas.

Por otra parte, es necesario controlar el acceso por parte de aplicaciones procedentes de otros lugares a recursos de la red.

Los problemas del control de acceso a ficheros y a la red constituyen los dos aspectos básicos del problema del confinamiento, tratado en anteriores capítulos, por lo que resulta fundamental su control.

Su resolución no es trivial, y a lo largo de las distintas versiones, tanto de la propia plataforma Java (distribuciones JDK de la empresa Sun), como de los diferentes navegadores que incorporen la máquina virtual de Java, han ido apareciendo distintas soluciones. En los siguientes apartados se analizará la evolución que ha ido sufriendo la seguridad de la plataforma Java a lo largo de su corta vida.

La seguridad que ofrece la plataforma Java ha ido evolucionando hacia un modelo cada vez más flexible, intentando consensuar la necesidad de protección con la de apertura, de forma que las posibilidades del usuario fueran lo más amplias posibles.

8.5 Modelo inicial de seguridad en Java

Los problemas de seguridad detectados inicialmente surgen con la aparición de los *applets*. La capacidad de cargar clases remotas proporciona una posible entrada para Caballos de Troya. El código cargado en una máquina puede haber sido escrito por usuarios malintencionados que intentan corromper ficheros locales o acceder a información confidencial en un nodo remoto. Así pues, el acceso al sistema de ficheros local debe ser controlado por un esquema de protección.

8.5.1 Política de seguridad: Caja de arena

Inicialmente el modelo de seguridad proporcionado por Java es el denominado modelo de la *“caja de arena”* (*sandbox*). Proporciona un entorno muy restrictivo para ejecutar el código no fiable (los *applets*) procedente de la red.

La esencia del modelo de caja de arena es que el código local es fiable, y por tanto tienen total acceso a recursos del sistema mientras que el código remoto (los *applets*) no es fiable y sólo pueden acceder a ciertos recursos, que conforman la denominada caja de arena. Así, por ejemplo, la caja de arena define un directorio del disco en el que la aplicación puede leer y escribir. El resto del sistema de ficheros permanece inaccesible para ésta. Este modelo se ilustra en la siguiente figura:

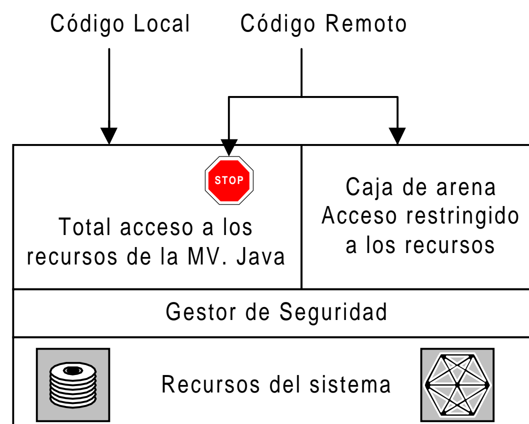


Figura 8.1: Modelo de la caja de arena, integrado en el kit de desarrollo de Java 1.0 (JDK 1.0).

8.5.1.1 Mecanismo de protección: Clase SecurityManager

Dado que el código local y remoto tienen que coexistir, el sistema debe saber si una llamada a un recurso crítico, como el disco, es realizada por código local o remoto. Para resolver este problema la máquina de Java trabaja de la siguiente manera:

Todas las clases que provienen de la red son cargadas por un cargador *ClassLoader*. Las clases procedentes del sistema de ficheros local tienen un cargador especial del sistema. Cada entrada en la pila de llamadas incluye una referencia a la clase que se ejecuta en esa entrada y el nombre del cargador de dicha clase. De esta manera se distinguen las clases locales de las remotas.

Para forzar a cumplir estas políticas, todos los métodos potencialmente peligrosos en el sistema tienen que ser diseñados para llamar a la clase *SecurityManager*. Tanto el JDK como HotJava y Netscape lo hacen así para implementar esta política de seguridad.

La clase *SecurityManager* implementa la política de la caja de arena, y comprueba si se permite la acción requerida, lanzando una excepción si se encuentra código remoto en la pila de llamadas. La clase *SecurityManager* implementa por tanto, un monitor de referencia.

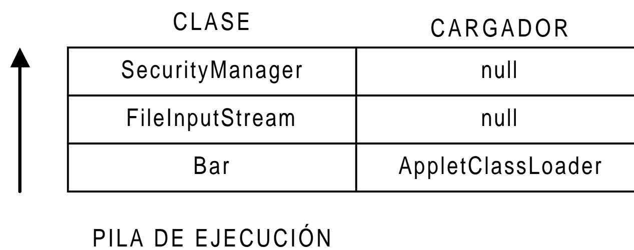


Figura 8.2: Pila de ejecución usada en la plataforma Java

Los gestores de seguridad en JDK y Netscape utilizan, pues, el contenido de la pila de llamadas para decidir si garantizar o no el acceso. El sistema de seguridad busca código remoto examinando la pila de llamadas (introspección en la pila, *stack introspection*). Si existe un cargador de clase diferente del cargador local (denotando así código remoto), entonces se aplica una política de código no fiable y se lanza la excepción. En otro caso se utiliza una política para código local de confianza (acceso sin restricciones).

El gestor de seguridad está protegido por el propio sistema de tipos del lenguaje Java que en teoría asegura que sólo se puede acceder al gestor de seguridad por los métodos que se definen al efecto.

8.5.2 Ampliaciones al modelo original: Firmas digitales (JDK 1.1)

La versión 1.1 del JDK, introduce el concepto de firma digital. Los programadores de clases Java que van a poder moverse por la red (*applets*) firman este código con su nombre, de forma que las empresas de software con reconocido prestigio ofrecen al cliente la seguridad de que el software firmado por ellos tiene garantías de que no realizará operaciones que perjudiquen la información del cliente. El control de acceso está, por tanto, basado en la fuente del código, y no en quién lo está ejecutando como ocurre en los sistemas tradicionales.

8.5.2.1 Firma digital

El funcionamiento de una firma digital es el siguiente:

1. Se firma un documento usando una clave privada, que puede ser generada por una herramienta especial o a través de una API de seguridad.

2. Se envía el documento firmado a otra persona.
3. Se le proporciona al receptor una clave pública, correspondiente a la clave privada usada para generar la firma.
4. El receptor utiliza la clave pública para verificar la autenticidad de la firma y la integridad del documento.

8.5.2.2 Política de seguridad de la caja de arena ampliada con firmas digitales

Para implementar la política de seguridad Java necesita la noción de autoridad o entidad a la que se reconocen unos derechos determinados en el sistema. En un sistema operativo tradicional cada usuario es una autoridad. En la plataforma Java cada firma es una autoridad. El código firmado se asocia con una autoridad. Cuando se verifica una firma, se une la autoridad que representa al objeto clase en la máquina virtual de Java. Cualquier código puede posteriormente preguntar a una clase por su autoridad.

Esto permite ampliar la política de la caja de arena, haciendo que las clases provenientes de autoridades en las que se confía tengan los mismos permisos que una clase local (acceso total). Cuando se desea que una clase remota tenga más privilegios de los que le ofrece la caja de arena, se añade la dirección URL de donde procede la clase, junto con su clave pública. Esto permitirá actuar a todas las clases remotas provenientes de esa autoridad (y esa dirección de red) como si fueran clases locales, y por tanto tener acceso a todos los recursos del sistema.

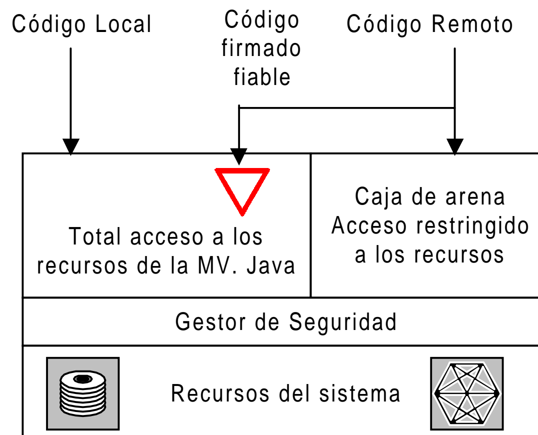


Figura 8.3: Modelo de seguridad integrado en JDK 1.1

8.5.2.3 Mecanismo de protección

Cuando se ejecuta una operación comprometida dentro de una clase procedente de la red, se comprueba si esta clase es considerada "de confianza" por parte del cliente. Para ello utiliza la clave pública almacenada en la política de seguridad y comprueba que la firma es auténtica. Una vez hecho esto, la clase se considera de confianza, y pasa a ser tratada como cualquier clase local, es decir con total acceso a los recursos.

Así pues, un *applet* firmado digitalmente se trata como si fuera código local fiable si la firma es reconocida como fiable por el sistema que recibe el *applet*.

8.5.3 Nueva arquitectura de seguridad en Java2 (JDK 1.2)

Aunque el modelo anterior amplía las posibilidades de trabajo, sigue siendo sin embargo, un mecanismo poco flexible y no exento de problemas. De nuevo se incumple el principio de mínimo privilegio, asignando más autoridad de la estrictamente necesaria.

La siguiente versión 1.2 del JDK[1] mejora considerablemente el modelo de seguridad ofrecido, incorporando una política más flexible, que se adapta a las necesidades de acceso de cada clase.

8.5.3.1 Política de seguridad

En primer lugar, todo el código, con independencia de ser local o remoto, puede estar sujeto a una política de seguridad. La política de seguridad define el conjunto de permisos disponibles para el código (determinado por su localización y su firma). Esta política puede ser configurada por el usuario o administrador del sistema.

Cada permiso especifica el acceso permitido a un recurso particular, tal como lectura y escritura a un fichero o conexión a una dirección remota de la red.

Para ello dispone de una clase *java.security.Policy* y a través de ella define la política de seguridad deseada. Esta clase representa en tiempo de ejecución el contenido de un fichero de políticas definido con un lenguaje específico, como se ve en el siguiente ejemplo. Se trata de definir los permisos de acceso que tendrán las clases en función del servidor del que se originan y/o de la firma que las acompañe:

```
grant CodeBase
     "http://www.uniovi.es/~fondon/", SignedBy "*"
     {     permission java.io.FilePermission "read,write", "/tmp/*";
           permission java.net.SocketPermission "connect", "*.uniovi.es";
     };
```

Con este código se está definiendo una política que implica que cualquier clase cargada desde la dirección http:// www.uniovi.es/~fondon/, firmada o no, tiene permiso para leer y escribir cualquier fichero situado en el directorio /tmp, y para conectarse a cualquier servidor dentro del dominio DNS uniovi.es.

El usuario o administrador de la máquina incluirá los permisos adecuados en la política de seguridad definiendo de manera apropiada este fichero.

8.5.3.2 Dominios de protección

El sistema en tiempo de ejecución (*runtime*) organiza el código en dominios. Cada dominio se asocia a un conjunto de clases cuyas instancias tienen concedido el mismo conjunto de permisos.

Un dominio de protección está formado por un conjunto de objetos, con sus permisos, que serán accesibles por aquellas entidades (objetos) con autorización, que tengan asociado ese dominio.

Un dominio puede constituir una caja de arena, si los permisos que ofrece son los mismos que ésta. El sistema contará con una serie de dominios que filtrarán el acceso a los objetos en distintos grados.

[1] Que se ha venido a llamar Java2.

Creación de dominios

Los permisos, establecidos en la política, se conceden a los dominios de protección, que se forman a partir de las reglas establecidas en la política.

Si la política dice que la clase A tiene acceso a los objetos X, Y y Z, cuando se cargue la clase A, se creará un dominio con esos objetos y sus permisos, y se le asignará ese dominio a la clase A.

En un ejemplo más concreto. Cuando se carga una clase, por ejemplo http://www.uniovi.es/~fondon/demo.class, el cargador consulta la política (el objeto política), y mira si la clase a cargar cumple los requisitos de alguna de las definiciones existentes. En caso de que así sea, crea un dominio de protección y le otorga los permisos que vengan definidos a través de la política. Si posteriormente se producen nuevas cargas de otras clases, si coinciden con el mismo patrón que el dominio establecido, se harán pertenecer a ese dominio y si no se creará un nuevo dominio para ellas.

Si una clase encaja en más de una definición en la política, entonces se sumarán los permisos y se creará un dominio con la unión de todos los permisos.

Así pues, cada clase pertenece a un único dominio, el cual dispone de una serie de permisos.

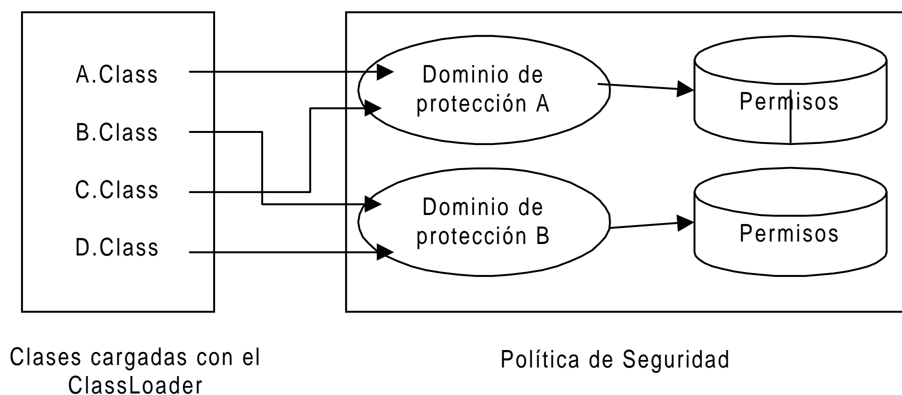


Figura 8.4: Asociación de clases en dominios de protección

La asignación de una clase a su dominio se registra cuando se carga la clase. Esta correspondencia se realiza llamando al método de la clase ProtectionDomain setProtectionDomain(Class, ProtectionDomain). Antes de cargar la clase, se comprueba que ésta puede ser cargada y se crea el dominio para ella, si es que aún no existe. Todas estas tareas las realiza el sistema cuando se encuentra con la definición de una clase que no está cargada.

Dominio del sistema

Todo el conjunto de clases que maneja el sistema forma parte del dominio del sistema. Se trata de un dominio de protección especial que tiene permiso de acceso a cualquier recurso.

Dominios locales

La política de seguridad se aplica también a las aplicaciones encontradas en el sistema de ficheros local. Para ello, se proporciona la clase java.security.Main, que puede ser usada para llamar a una aplicación local.

Este uso asegura que cualquier aplicación local que se encuentre dentro de los directorios definidos en java.app.classpath se cargue con el SecureClassLoader y esté sujeta a la política de seguridad. Se crearán dominios de protección separados para tales aplicaciones.

8.5.3.3 Permisos de acceso

Cuando se crean los dominios, se crean los permisos que esos dominios tendrán asociados. La existencia de un permiso determinado dentro de un dominio tiene el significado de que las clases que pertenecen al dominio están autorizadas a realizar las operaciones sobre los recursos que por convenio se otorguen al permiso.

Existe una jerarquía de clases que sirven para almacenar los permisos de acceso. Todas las clases permiso que se creen deberán heredar de dos tipos de clases raíz: la clase java.security.Permission o la clase java.security.BasicPermission.

La diferencia entre ambas es que la primera define permisos más complejos, que requieren nombres y acciones. El nombre suele indicar un recurso individual dentro del conjunto de recursos a proteger con ese permiso. Las acciones indican las operaciones que permite la posesión del permiso sobre el recurso. Por ejemplo, la clase java.io.FilePermission requiere un nombre de fichero (por ejemplo "fichero1") y las acciones de lectura, escritura, etc. (por ejemplo "read, write"). La existencia de este permiso en el dominio significaría que se permite el acceso de lectura-escritura al fichero "fichero1".

La clase BasicPermission define permisos más simples, que sólo requieren un nombre. Por ejemplo, java.lang.RuntimePermission hereda de la clase java.security.BasicPermission y simplemente necesita un nombre (como "exitVM", que permite a los programas salir de la máquina virtual Java).

Cada paquete de clases tiene que implementar su propia clase de permisos, adaptados a los permisos que necesita si quiere aprovecharse de la infraestructura proporcionada con este sistema de seguridad.

8.5.3.4 Control de acceso a recursos

El control de acceso se realiza en aquellas operaciones de acceso a recursos que se consideren peligrosas y por tanto que necesitan que su acceso se restrinja a ciertos objetos. Cualquier programador puede incluir control de acceso en las clases que implementen recursos que se necesiten proteger. Como ejemplo, clases del sistema tales como la parte de acceso al sistema de ficheros también se controlan de esta manera (para lo cual se han tenido que reescribir en parte para la distribución JDK 1.2).

Cuando se implementa una clase, si se desea controlar el acceso de otros objetos sobre ella, dentro de la implementación de la clase servidora, el programador deberá incluir una serie de llamadas al gestor de seguridad para que compruebe si el acceso debe permitirse, a través del objeto controlador *AccessController*. En concreto, al recibir la operación, se crea un permiso igual al que la clase cliente debería tener para poder realizar la operación, y se le pide al controlador que determine si el permiso se encuentra dentro de los permisos del dominio[1]:

```
FilePermission p = new FilePermission("fichero1", "read");
AccessController.checkPermission(p);
```

[1] La semántica con que se interpreta el permiso es totalmente arbitraria y corresponde al implementador de la clase servidora.

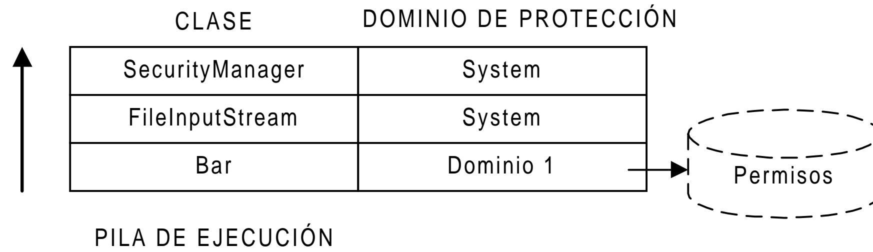


Figura 8.5: Pila de ejecución con dominios asociados

En el caso de que sólo haya un dominio involucrado, la comprobación que realiza este objeto consiste en saber si la operación requerida está incluida entre los permisos que contiene el dominio de protección actual. Examinando las entradas en la pila de llamadas del sistema (*stack introspection*) éste va determinando los dominios asociados a cada clase de la pila de llamadas. Consultando la política de seguridad se obtienen los permisos que el dominio (cliente) tiene sobre el objeto en cuestión y comprueba si la petición es sobre un objeto del mismo dominio, y en ese caso mira si el permiso que se necesita está dentro de los permisos que contiene el dominio. Si no es así, se lanza una excepción de seguridad.

En el caso de que en la pila de llamadas se encuentren clases de diferentes dominios de protección (una clase A con un dominio A llama a una clase B con un dominio B diferente, etc.) el permiso debe aparecer en todos y cada uno de los dominios involucrados (para estar acorde con el principio del mínimo privilegio). Es decir, se tiene en cuenta la intersección de los permisos de todos los dominios.

Figura 8.6: Pila de ejecución con llamadas de diferentes dominios

Ampliación de privilegios

En el ejemplo anterior, en la pila de llamadas se están produciendo cambios de dominio al invocar a objetos que pertenecen a dominios diferentes del de la clase que llama (por ejemplo una clase que lee un fichero mediante FileInputStream, que pertenece al dominio del sistema). Para mantener el principio de mínimo privilegio, es necesario que un dominio no pueda ganar permisos adicionales como resultado de llamar a un dominio más "poderoso", mientras que un dominio más poderoso debe perder su poder cuando llama a un dominio con menos poder.

Para ello, los permisos disponibles para un hilo en ejecución son la intersección de los permisos de todos los dominios de protección por los que ha pasado hasta el momento.

Sin embargo, en algunos casos es necesario que una clase trabaje con sus propios permisos, en lugar de con los permisos normales que tendría como anteriormente. Un ejemplo es el de una clase del sistema que tiene que actualizar la fecha de un fichero aunque la clase cliente sólo tenga permiso de lectura sobre el mismo. Con el mecanismo anterior esto no podría realizarse ya que la clase del sistema funcionaría con los privilegios anteriores, más restringidos.

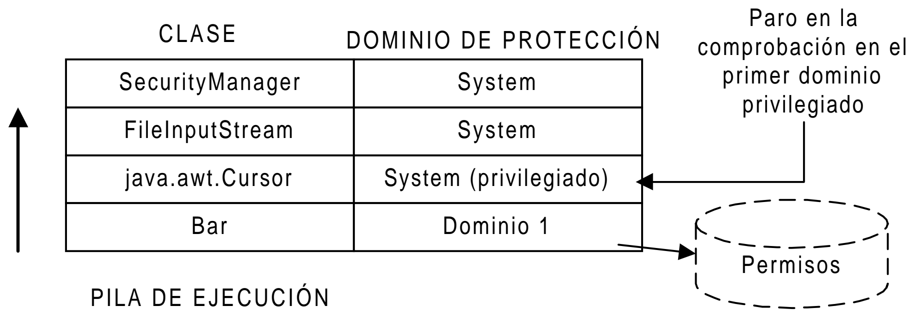


Figura 8.7: Pila de ejecución con llamadas a dominios privilegiados

Para permitir esto, se dispone de dos primitivas en AccessController: beginPrivileged() y endPrivileged(). Insertando la primera en el código de una clase se indica al soporte en tiempo de ejecución que se utilicen los propios permisos de la clase en lugar de la intersección de los permisos de la cadena de llamadas. Con la segunda se indica el fin de este modo de trabajo.

Este mecanismo de funcionamiento es análogo al del bit *setuid* del Unix, y por tanto adolece de sus mismos inconvenientes.

8.5.3.5 Resumen de la integración de seguridad en el JDK1.2

El sistema cuenta con un mecanismo para definir políticas. En ellas se definen los permisos de acceso que pueden tener las clases en función de dos parámetros: el lugar de procedencia de la clase y la firma digital que posea. Los usuarios o administradores son los que definen las políticas que se quieren aplicar. Todo esto se combina con los procedimientos ya existentes de verificación de código y de firmas digitales para las clases remotas.

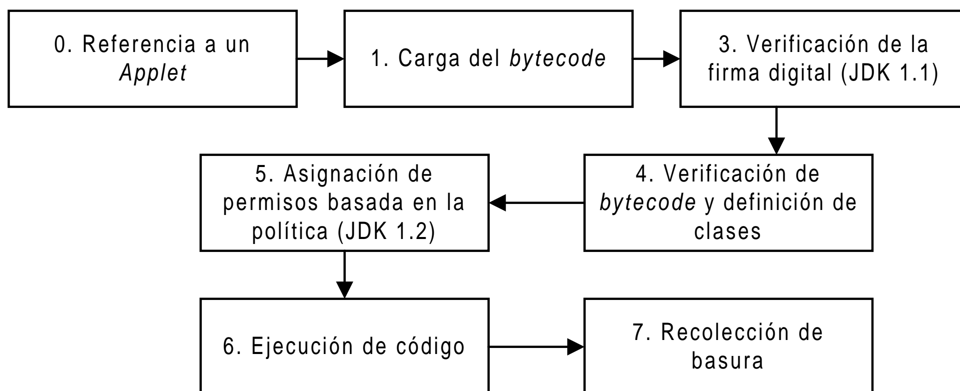


Figura 8.8: Integración de seguridad en el JDK 1.2

Cuando se carga una clase, se le asocia un dominio, que indica el conjunto de permisos que tiene dicha clase a su disposición. El dominio asociado dependerá de la especificación hecha en la política. Varios objetos pueden tener el mismo dominio asociado si tienen los mismos permisos.

Para ejercitar el control de acceso si se desea que el acceso a una clase no sea libre, si no que se restrinja en función de la política diseñada, será necesario que en la implementación de esa clase se codifiquen operaciones relativas a la seguridad. Para ello, en la implementación de la clase se crearán los permisos que se consideren necesarios para desarrollar cada operación en cuestión. Estos permisos se pasarán al gestor de seguridad (controlador de acceso), que comprobará que el dominio que invoca a la clase servidora posee esos permisos examinando los dominios asociados a cada clase de la pila de llamadas (introspección en pila).

El gestor de seguridad que comprueba permisos tendrá en cuenta lo siguiente. Si el objeto cliente y el servidor están en el mismo dominio, simplemente comprueba que la petición esté entre los permisos asociados al dominio. Si, por el contrario, no pertenecen al mismo dominio, entonces calcula un "dominio actual", que será la intersección de todos los dominios por los que haya pasado el hilo de ejecución (para mantener el principio del mínimo privilegio). Posteriormente se comprueba si el permiso solicitado se encuentra dentro de este dominio.

Existe un caso especial, en el que el programador quiere explícitamente que no se aplique el mecanismo anterior y que se utilicen los permisos de la propia clase efectiva en lugar de la intersección de los permisos de la pila de llamadas, con un efecto análogo al del bit *setuid* del Unix (amplificación de privilegios). Para ello deberán introducirse en el código las primitivas de beginPrivilege y EndPrivilege que delimitan la parte del programa en la que se quiere llevar a cabo esta amplificación (temporal) de privilegios.

8.6 Crítica a la seguridad en Java

8.6.1 Inconvenientes de la caja de arena

8.6.1.1 Inflexibilidad por la granularidad gruesa y falta del principio de mínimo privilegio

El modelo de seguridad de caja de arena resulta a todos visos muy inflexible. Todas las clases remotas tienen los mismos permisos de acceso, y no es posible ajustar la protección a las necesidades de cada aplicación. Es un modelo de todo o nada, por lo que, en la mayoría de las ocasiones, o bien se conceden más permisos de acceso de los necesarios o bien se conceden menos de los que serían deseables.

Dividir el código en código local totalmente confiable y código remoto potencialmente peligroso y por tanto incapacitado para realizar prácticamente cualquier tarea resta mucha utilidad al sistema. Por un lado se impide la utilización de código remoto que pudiera realizar tareas de utilidad en recursos no vitales del sistema (como formatear un fichero en pantalla), puesto que para ello habría que darle acceso a todos los recursos (sacarlo de la caja de arena) y podría utilizarlos con malevolencia. Por otra parte el código local tiene acceso a todos los recursos, aunque no lo necesite para realizar su función y se está expuesto a fallos en la implementación de las aplicaciones locales que comprometan el sistema.

Lo anterior es un ejemplo de que la caja de arena incumple el principio de mínimo privilegio, por lo grueso de la granularidad de protección que utiliza: o se tienen todos los permisos o no se tiene (prácticamente) ninguno.

8.6.2 Inconvenientes de la protección basada en firmas digitales

8.6.2.1 Suposición de buena voluntad de los proveedores

Las firmas digitales parten de que el usuario debe suponer que los proveedores de software que firman las aplicaciones que recibe tienen buena voluntad, y que sus aplicaciones van a funcionar cómo ellos indican, sin realizar nada malévolo en el sistema. La existencia de entidades certificadoras de las firmas simplemente significa que (en teoría) se puede determinar quién es el proveedor de una aplicación determinada, no que la aplicación no funciona de manera anómala o destructiva.

8.6.2.2 No es un mecanismo de protección

Por tanto, en realidad este mecanismo no proporciona protección al usuario en absoluto. El sistema no se protege evitando accesos erróneos o malintencionados, simplemente permite saber a quién se le puede echar la culpa de estos problemas, a través de la firma digital de la clase problemática. Nunca se puede evitar este daño potencial pues es imposible asegurar que por muy fiable que sea la fuente el código esté libre de problemas.

Todo lo más, se ofrece cierta tranquilidad psicológica al usuario al permitirle conocer quién origina el código que está ejecutando y decidir si confía o no en la fuente. En cuanto a lo que es el modelo de protección, éste sigue siendo en esencia el mismo que el de la caja de arena, con todos sus problemas.

8.6.2.3 Limitación de la expansión del mercado de software

Sin embargo, esta supuesta protección psicológica puede imponer una restricción en la expansión del mercado del software, al limitar la existencia de pequeños proveedores de software. Al no garantizarse protección y basarse todo en la confianza del proveedor, los usuarios inconscientemente pueden tender a adquirir y utilizar software de casas con más reputación (o simplemente más famosas), y descartar de antemano software de pequeñas casas menos conocidas. De hecho, este es un argumento comercial que las casas más poderosas pueden utilizar para imponer sus productos frente a productos de casas con menos influencia (que no necesariamente son peores).

8.6.3 Inconvenientes de la plataforma Java con la nueva arquitectura

8.6.3.1 Base de computación fiable muy grande y compleja

El conjunto de código del sistema que se encarga de las tareas relacionadas con la seguridad/protección (base de computación fiable) es muy grande. Además, el sistema de seguridad es muy complejo, interactuando muchas piezas diferentes: cargador de clases, verificador de *bytecode*, elementos de la nueva arquitectura de seguridad, etc. No existe una economía de mecanismos. Esto hace que la probabilidad de que existan agujeros en la implementación de la seguridad en la plataforma es muy alta, y de hecho existen muchos [Kim97, DFW96].

Con la nueva arquitectura, cada clase se asigna a un dominio de protección en función del sitio origen y la firma digital que porta. Sin embargo, todas las clases que están en el mismo dominio comparten todos los permisos del dominio. Es decir, todas las clases de un dominio tienen los mismos privilegios, aunque no los necesiten todos.

Esto mejora la granularidad de protección de la caja de arena, aunque todavía no se consigue el principio de mínimo privilegio, que consistiría en dar a cada clase individual exactamente los permisos que necesitase. Para ello sería necesario otorgar a cada clase un dominio propio. La configuración del sistema en su conjunto impide actualmente esto, y la existencia de tantos dominios de protección en el sistema posiblemente enlenteciera el mismo en demasía.

8.6.3.2 Granularidad no al nivel de objetos individuales

El sistema de protección asigna cada clase a un dominio. La protección es al nivel de clases: todos los objetos de la misma clase son equivalentes en cuanto a los permisos de acceso que tienen (los permisos de su clase). Esto hace que no sea posible establecer diferencias en la protección con respecto a objetos individuales, si es que estos pertenecen a la misma clase. No es difícil imaginar situaciones en las que esto sea necesario, por ejemplo, que dos instancias diferentes de la misma clase “Persona” sean tratadas de diferente manera

cuando llamen al método “entrar” de la clase “Portero_Discoteca”. Esto es un aspecto más del principio de mínimo privilegio que no se cumple.

8.6.3.3 Falta de uniformidad y generalidad del mecanismo: responsabilidad del usuario en la implementación de la protección

No hay uniformidad en el tratamiento de los objetos (clases). La programación de clases cliente se realiza de distinta manera de las clases servidoras. Es necesario codificar expresamente en las clases servidoras si queremos que éstas sean protegidas. Además, esta codificación puede conllevar bastante código y ser relativamente compleja (creación de clases permiso y alguno de sus métodos, uso del controlador de acceso, etc.). Al dejar en manos del usuario parte del sistema de protección, la posibilidad de introducción de errores en estas partes que provoquen problemas de seguridad es elevada.

8.6.3.4 Falta de flexibilidad para introducir protección a posteriori

Por otro lado, si una vez codificada una clase sin previsiones para que sea protegida, aparece posteriormente esta necesidad, no se puede llevar a cabo sin disponer del código fuente original de la clase. En general no puede asegurarse que se disponga del código fuente de todas las clases que sea necesario utilizar en una aplicación. Por ejemplo, las propias clases de la distribución de Java (JDK) se tuvieron que modificar para protegerlas con este esquema [GS98].

8.6.3.5 Falta de adaptabilidad. Mecanismo “pesado”

La complejidad del mecanismo y el elevado número de elementos que lo componen hacen que este sea un mecanismo “pesado”, y el sistema sea difícil de adaptar para situaciones específicas. El problema es que el sistema debe llevar esta estructura completa en todas sus instalaciones, necesítese o no esta estructura. Por ejemplo, en sistemas empotrados que no necesiten seguridad se debe pagar el precio de esta estructura aunque no se use. Como mínimo el código de las clases del sistema utiliza esta estructura de seguridad, con lo que, además del incremento en el tamaño del sistema se disminuye el rendimiento del mismo.

8.6.3.6 Sin mediación total

No se cumple el principio de mediación total, ya que el sistema de protección sólo interviene en ciertas llamadas que se decide arbitrariamente que son importantes, cuando el programador de las clases pide explícitamente que se compruebe si el cliente tiene los permisos adecuados. El sistema de protección no media en absolutamente todas las llamadas, como requiere el principio de mediación total.

8.6.3.7 Aceptación difícil por el usuario

El mecanismo es lo suficientemente complicado para que muchos programadores no lo utilicen en la práctica y desarrollen sus clases sin previsiones para la protección. Si a esto unimos la falta de flexibilidad para incorporar la protección a posteriori cuando se necesite utilizar estas clases en sistemas que necesiten protección, se limita la posibilidad de desarrollar aplicaciones más seguras que las actuales.

9 Elección de Capacidades como Mecanismo Base de Protección

El diseño realizado del núcleo de seguridad para un sistema integral orientado a objetos puede describirse agrupando éste en tres aspectos básicos:

- Elección del Modelo de protección. En primer lugar se decide el modelo de protección que se considera más adecuado en función de los requisitos impuestos.
- Diseño del Modelo. Una vez elegido el modelo, se detallan las características del diseño de este modelo, para su implantación en un sistema integral orientado a objetos.
- Diseño del Mecanismo de Protección. Por último, se describen los aspectos de diseño del mecanismo, adecuado a las características del modelo y del sistema en el que se implanta.

En este capítulo se justifican las razones por las que se toma la decisión de uso de capacidades como modelo de protección. En capítulos siguientes se realiza el desarrollo de las del diseño del modelo y el mecanismo, y se analizan las razones por las que se ha llegado a este diseño particular.

9.1 Análisis del uso de listas de control de acceso

Como se ha explicado en el capítulo 4, algunos sistemas utilizan listas de control de acceso (LCA) como mecanismo de protección. Cada elemento a proteger posee una lista con los sujetos o dominios que tienen acceso sobre una o varias de sus operaciones. Puede pensarse que las listas de control de acceso funcionan como un portero de un edificio que consulta con una lista de clientes autorizados cada vez que un cliente intenta entrar en el edificio.

El uso más común corresponde a sistemas operativos que usan LCA asociadas a ficheros, en las que el dominio de protección está asociado al usuario. Cualquier proceso que ejecute un usuario tendrá acceso a todos los recursos sobre los que éste tenga autoridad. La siguiente figura pretende describir la situación.

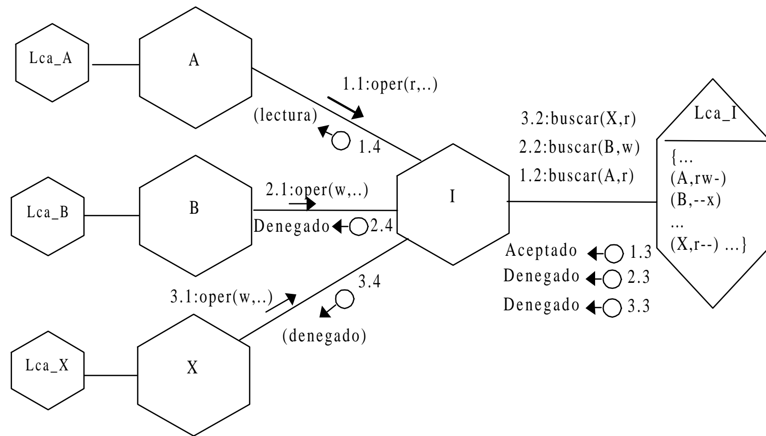


Figura 9.1: Funcionamiento de las listas de control de acceso

Sin embargo las listas de control de acceso tienen algunos inconvenientes importantes a la hora de implantarlas en un sistema integral orientado a objetos: problemas de escalabilidad, uniformidad, granularidad fina, flexibilidad, sobrecarga, dependencia entre objetos y complejidad, los cuales se analizarán a continuación.

9.1.1 Falta de escalabilidad

La propia concepción de las listas de control de acceso tiene graves problemas de escalabilidad en diferentes ámbitos, como el número de objetos de un sistema y su movilidad.

9.1.1.1 Sistemas con gran número de objetos: problemas de escalabilidad en el espacio y en el tiempo

En un sistema que utilice listas de control de acceso, cada objeto a proteger lleva asociada una lista con los objetos que pueden acceder a éste junto con sus permisos. Dado que un SIOO constituye un mundo de objetos de cualquier granularidad, el tamaño de las listas y la cantidad de ellas constituiría un problema de escalabilidad importante cuando los sistemas tienen un número de objetos realista.

Existe un problema de escalabilidad del espacio de almacenamiento de la lista, que aumenta a medida que aumenta el número de objetos del sistema.

Otro problema de escalabilidad derivado es el del tiempo (sobrecarga) de comprobación de permisos. El tamaño de una lista de un objeto aumenta a medida que aumenta el número de objetos que pueden acceder a él. Puesto que en el acceso hay que comprobar los permisos buscando en la lista, el tiempo de comprobación no es constante en general y varía con el tamaño de la lista. Cuanto mayor sea la lista, mayor será este tiempo de comprobación.

Puede reducirse este tiempo usando estructuras avanzadas de almacenamiento de la LCA, lo que normalmente se traduce en un mayor espacio de almacenamiento necesario y perjudicando aún más la escalabilidad en el espacio.

Por otro lado, en un sistema distribuido es muy difícil conocer por anticipado qué objetos tendrán necesidad de acceder a un objeto determinado, información necesaria para poder construir la LCA.

9.1.1.2 Problemas de escalabilidad en la movilidad de objetos

Esto que de por sí las hace inadecuadas, se hace aún más inconveniente cuando entra en juego la posibilidad de migración (movilidad) de objetos. En este caso, además de migrar el objeto es necesario migrar su lista de control de acceso asociada, lo cual hace que la operación de migración tenga una sobrecarga adicional.

9.1.2 Intento de reducción de los problemas de escalabilidad

9.1.2.1 Reducción del número de LCA: agrupación en ámbitos

Para reducir estos problemas de escalabilidad, los sistemas operativos que utilizan LCA no lo hacen al nivel de objetos como entidades absolutamente independientes, sino que éstos se agrupan en ámbitos, formados por grupos de objetos que comparten el mismo espacio de direcciones (normalmente denominados procesos) y por tanto se suponen confiables, como ocurre en Guide [Hag94], Amadeus [CBH+92] o Choices [CM90].

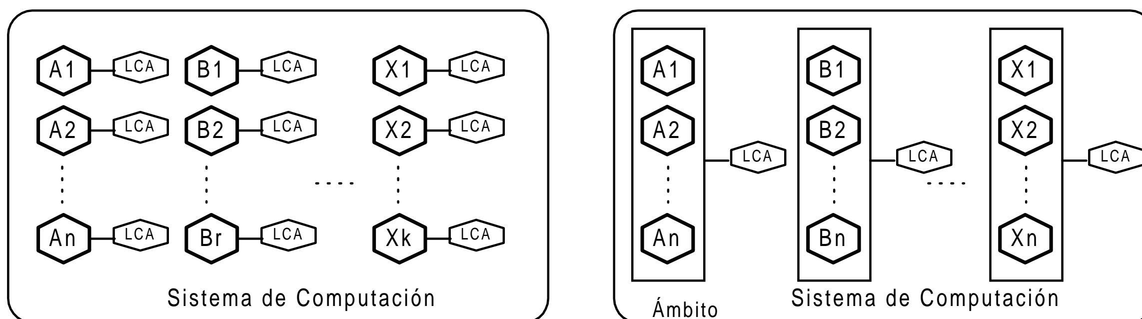


Figura 9.3: Reducción del número de LCAs

Este hecho reduce el número de listas de control que existen en el sistema, aunque hace que aparezca el problema de la mediación parcial al aumentar la granularidad. Por otro lado, para otro tipo de recursos, como los ficheros, se sigue necesitando una protección individual de cada recurso, y no admiten agrupamientos de este estilo.

9.1.2.2 Reducción del número de entradas de cada LCA: usuarios y grupos

Por otra parte el dominio de protección suele ser determinado por el usuario. Para reducir el número de entradas en cada LCA, en lugar de una entrada por objeto se establece una entrada por cada usuario (denota un conjunto de objetos grande pertenecientes al mismo). Para reducirlo aún más, se crean grupos de usuarios y se hacen entradas por grupos de usuarios. Las LCA para proteger el acceso a ficheros en Unix llevan esto al extremo de tener sólo tres entradas: una para los permisos del usuario propietario del fichero, otra para los de un grupo de usuarios determinado que se defina y otra para el resto de los usuarios.

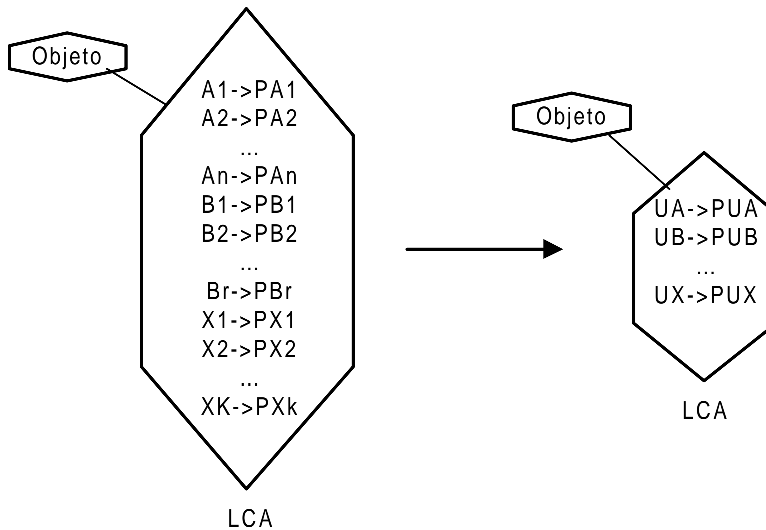


Figura 9.4: Reducción del número de entradas en la LCA

Sin embargo, esto produce problemas adicionales. De hecho, al reducir el tamaño de las listas de control de acceso lo que se está haciendo es aumentando el tamaño de la granularidad de protección puesto que ya no se pueden indicar permisos individuales para objetos individuales, si no que son compartidos por conjuntos grandes de objetos denotados por el usuario, grupo, etc. Esto impide seguir el principio de mínimo privilegio.

9.1.3 Problemas adicionales debidos a la falta de escalabilidad

Las soluciones que se introducen para evitar el problema de escalabilidad en las listas de control de acceso provocan a su vez problemas adicionales de uniformidad, granularidad fina y flexibilidad.

9.1.3.1 Falta de uniformidad

Como se acaba de mostrar, los sistemas que utilizan listas de control de acceso agrupan en general a los objetos en dominios o ámbitos que comparten los mismos permisos de acceso. Los objetos situados en el mismo dominio se encuentran en el mismo espacio de direcciones, por lo que su acceso está protegido por hardware frente a objetos de otros espacios. Para realizar una operación entre objetos de dominios diferentes, el mecanismo de control actúa mediante la comprobación en las listas de control de acceso. Sin embargo, los objetos del mismo ámbito no son protegidos entre sí, considerándose que la seguridad debe ser ejercida por el programador.

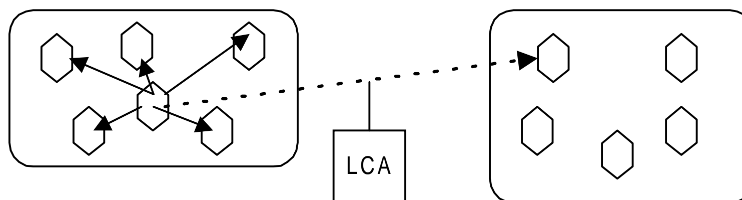


Figura 9.2: Uso de LCA en aquellas invocaciones que suponen cambio de dominio

El hecho de que las listas de control de acceso se utilicen para el acceso a un ámbito diferente, normalmente asociado a otro espacio de direcciones y en general a otro dominio, hace que la protección no sea tratada de forma uniforme para todos los objetos. Se produce una diferenciación en el control de acceso según el tipo de objetos al que se desee acceder.

9.1.3.2 Falta de granularidad fina: mediación parcial sin mínimo privilegio

Uno de los requisitos que se imponen en el diseño de la seguridad de un sistema integral orientado a objetos es la protección de granularidad fina, es decir, que la protección se realice en objetos de cualquier tamaño, y además debe permitir especificar permisos individuales para cada objeto del sistema sobre cualquier otro objeto. Como ya se comentó en el apartado anterior, los objetos situados dentro del mismo espacio de direcciones no se protegen con un mecanismo de listas de control de acceso.

Mediación parcial, no total

La protección de granularidad fina constituye un requisito impuesto en un sistema integral (véase el capítulo 6) y también está relacionado con uno de los principios básicos de diseño de un sistema de seguridad, la mediación total (véase el capítulo 3), por la que se impone la imposibilidad de esquivar el mecanismo de protección en cualquier tipo de acceso. En un buen diseño de seguridad cada acceso debe ser cotejado con la información de control de acceso, y no debe ser posible esquivar el mecanismo en ningún tipo de acceso. Si sólo se comprueban los permisos de acceso cuando se accede a objetos fuera del dominio actual, la mediación será parcial, y no total.

Incumplimiento del principio de mínimo privilegio

De la misma manera, la libertad existente dentro de un dominio hace que no se cumpla el principio de mínimo privilegio en el caso de los objetos situados dentro del mismo dominio de protección. Al no existir control, cualquier objeto disfruta de todos los permisos asociados a su dominio, que siempre serán más de los estrictamente necesarios para desarrollar su tarea.

La extensión del mecanismo de listas de control de acceso a objetos de cualquier granularidad resulta inviable por la sobrecarga que ello supondría, tanto en tiempo de comprobación, en cada invocación a objeto, como en ocupación por parte de las LCAs, que verían incrementado su tamaño de forma exponencial al tener que incluir un número de objetos muy elevado (puesto que antes sólo incluía ámbitos). El inconveniente de la sobrecarga será tratado posteriormente en un apartado específico.

9.1.3.3 Poca flexibilidad

La mayoría de los sistemas que utilizan listas de control de acceso trabajan al nivel de usuarios o incluso al nivel de grupos de usuarios en lugar de al nivel de objetos, es decir, la lista contiene una entrada por cada usuario o grupo de usuarios del sistema, con los permisos sobre el objeto. De este modo, el tamaño de la lista permanece controlado. Sin embargo ello significa la implicación del concepto de usuario en el mecanismo básico de protección. El concepto de usuario es un concepto de más alto nivel, que al ir unido de manera indisoluble al mecanismo de protección hace que éste tenga poca flexibilidad.

La existencia del concepto de usuario lleva aparejada la existencia de una serie de estructuras de datos y código que siempre debe estar presente en el sistema, aunque no sea realmente necesario, como en el caso de sistemas empotrados. Estos sistemas no necesitan el concepto de usuario, aunque sí la existencia de un mecanismo de protección.

9.1.4 Otros problemas de las listas de control de acceso

9.1.4.1 Sobrecarga del mecanismo de protección

La protección a través de listas de control de acceso implica la necesidad de realizar una búsqueda en la lista asociada al objeto cada vez que se intenta efectuar una invocación sobre alguna de sus operaciones. Este hecho resulta una carga considerable en el rendimiento del un sistema en el que se pretende establecer protección al nivel de objetos de cualquier granularidad, ya que realizar una búsqueda en una lista que potencialmente puede tener muchas entradas puede ser muy costoso.

Las estrategias anteriores para reducir el número de listas de control de acceso y el tamaño de las mismas también reducen como es lógico esta sobrecarga, a costa de los problemas descritos.

Comprobación sólo en el primer acceso: imposibilidad de revocación

Una estrategia adicional para reducir esta sobrecarga consiste en utilizar la lista de control de acceso únicamente la primera vez que se accede al objeto. Subsecuentes accesos se realizan ya directamente sin ninguna protección (y sin sobrecarga de búsqueda en la LCA). Como es evidente, esto va aún más en contra del principio de mediación total.

Por ejemplo, los sistemas operativos como Guide, Amadeus o Choices que utilizan listas de control de acceso restringen la protección a ciertos momentos concretos, y así sólo efectúan la comprobación la primera vez que se realiza una invocación del método de un objeto. Una vez comprobado, se establece un enlace que permitirá el acceso directo sin comprobación para el resto de los accesos al método.

Otro ejemplo es el mecanismo de protección de ficheros en Unix. La comprobación de la LCA de un fichero sólo se realiza una vez al abrir el fichero. Esta operación devuelve un manejador de fichero que permite a partir de ese momento realizar las operaciones sin volver a comprobar los permisos.

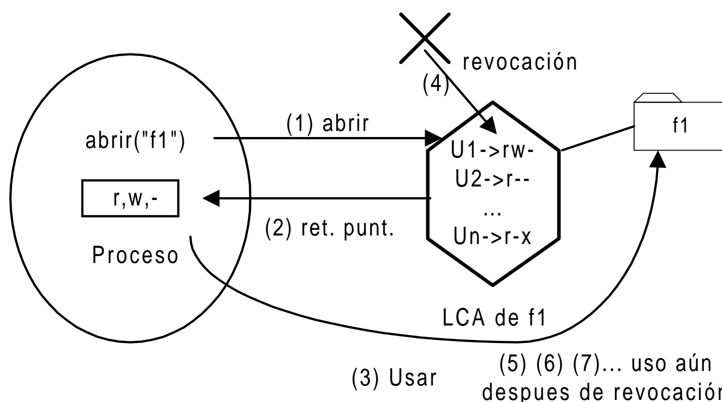


Figura 9.5: Imposibilidad de revocación debido a la comprobación sólo en la primera invocación

Este hecho, sin embargo, anula una de las características más destacables de las listas de control de acceso con respecto a las capacidades (al menos en teoría), que es la posibilidad de revocación. Si el enlace se produce en la primera llamada al método (o cuando se abre un fichero) y a partir de ahí permanece establecido ya no se podrá efectuar revocación de permisos sobre ese método hasta que el cliente no desconecte por propia voluntad el acceso que tiene al objeto (fichero).

9.1.4.2 Dependencia entre objetos

El uso de listas de control de acceso necesita de un conocimiento de quiénes son los objetos (cliente) que tienen permisos (pueden acceder) sobre el objeto (servidor) propietario de la lista. Se produce por tanto una dependencia entre objetos ya que el objeto servidor necesita conocer quiénes son los objetos cliente, que pueden acceder a él. En un sistema integral orientado a objetos, éstos deben ser entidades autosuficientes y absolutamente independientes entre sí, requisitos que chocan frontalmente con la dependencia proporcionada por las listas de control de acceso.

9.1.4.3 Complejidad

El modo de funcionamiento que necesita el mecanismo de listas de control de acceso necesita la existencia de una serie de estructuras y algoritmos adicionales. Estas estructuras y algoritmos son de una complejidad relativamente elevada teniendo en cuenta la sencillez conceptual inicial de la función que debe desempeñar un mecanismo de protección.

9.2 Análisis del uso de Capacidades

9.2.1 Definición de capacidades

El término capacidad fue introducido por Dennis y Van Horn en 1966. La idea consiste en un *token* que designa un objeto y da a un programa la autoridad para realizar un conjunto específico de acciones sobre él. Dicho de otra manera, una capacidad identifica de forma única un objeto y un conjunto de permisos de acceso a dicho objeto.

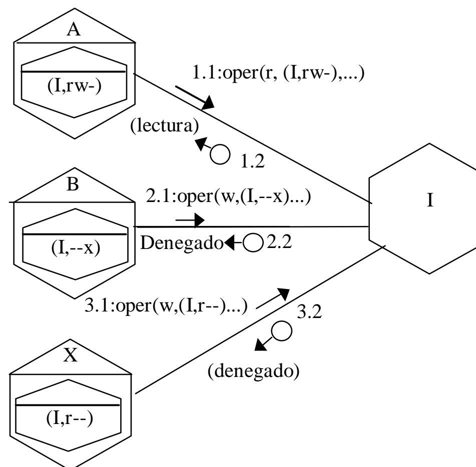


Figura 9.6: Funcionamiento del mecanismo de capacidades

Analogía con el mundo real

Una capacidad puede ser perfectamente comparable con una llave dentro de un llavero. Por ejemplo la llave de un coche [Shap99]. Esta llave designa a un objeto particular, en este caso un determinado coche, y cualquiera que posea la llave del coche podrá realizar ciertas operaciones con él (por ejemplo abrir y cerrar el coche, arrancarlo, abrir el maletero, etc.). Esta llave tan sólo permitirá operar en ese coche, y no en ningún otro. Por otro lado, el coche no sabe quién lo está utilizando, le basta con que se emplee la llave correspondiente. De forma análoga ocurre con el uso de capacidades, el objeto al que designa la capacidad no sabe quién lo usa, simplemente sabe que tiene una capacidad de acceso a él con los permisos adecuados.

Las llaves de los coches, en muchas ocasiones tienen variaciones. Es habitual tener, por ejemplo una llave para abrir el maletero, otra para arrancar y otra para abrir el depósito de gasolina. De la misma forma, dos capacidades pueden designar al mismo objeto, pero con diferente autorización. Un programa podría poseer una capacidad de sólo lectura para un fichero, mientras que otro podría tener capacidad para lectura y escritura sobre el mismo fichero.

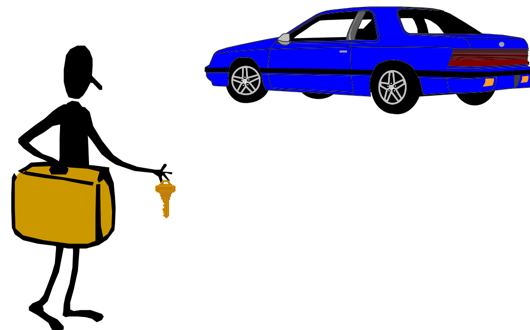


Figura 9.7: Analogía entre capacidad y llave

9.2.2 Fundamento de las capacidades

Se puede resumir en los siguientes puntos:

- Una capacidad identifica de forma única a un objeto y un conjunto de permisos de acceso sobre sus operaciones. Un objeto que posee una capacidad puede realizar las operaciones indicadas en los permisos de acceso del objeto denominado en la capacidad.
- La posesión de una capacidad es condición necesaria y suficiente para acceder al objeto asociado a la capacidad, y con la autoridad concedida por la capacidad. No existe otra forma de realizar operaciones sobre el objeto.

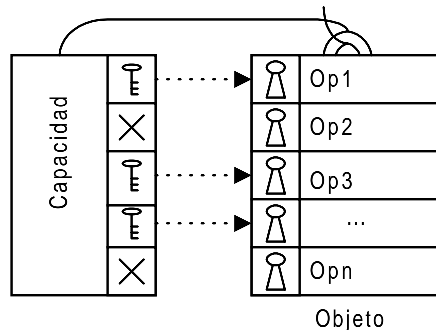


Figura 9.7 : Operaciones del objeto, emparejadas con los permisos de acceso: Analogía de llaves y cerraduras.

9.2.3 Control de granularidad fina

Un sistema que utilice capacidades con control de granularidad fina, deberá comprobar la existencia de los permisos adecuados en la capacidad en cada invocación, de forma independiente del objeto del que se trate. Este hecho, provocará una cierta sobrecarga del mecanismo, lo cual puede resultar excesivamente costoso. La mayoría de sistemas que utilizan capacidades, como Amoeba, no realizan este control de grano fino, puesto que no son sistemas integrales orientados a objetos, y sus objetos son entidades de granularidad gruesa.

Hay que hacer notar, sin embargo, que el principio de mediación total y de mínimo privilegio obligan en cualquier caso a realizar esta comprobación de permisos en todos los

accesos. Y por otro lado, la comprobación de permisos siempre lleva asociado un costo ineludible e independiente de que se utilice un mecanismo de control de acceso u otro.

9.2.3.1 Mejora de la escalabilidad en el espacio y en el tiempo

La posibilidad de poder ejercer este tipo de control supone sobrecarga, pero el uso de capacidades permite una mayor escalabilidad.

Por una parte, el uso de capacidades no supone un incremento considerable en la memoria que se necesita para su implementación. Intuitivamente, simplemente se necesita añadir la memoria necesaria para guardar los permisos que concede la capacidad[1]. En cuanto al tiempo, el costo de comprobación de permisos en la invocación a métodos es constante y no depende del número de objetos que existan en el sistema, si no de los permisos que conceda la capacidad. Por otro lado, el costo individual es menor al ser más sencillo en general comprobar una pequeña secuencia de permisos que hacer una búsqueda en una lista de control.

9.2.4 Independencia entre objetos respecto al mecanismo de protección

El uso de capacidades no obliga a la necesidad de conocimiento del entorno de objetos con el que se puede relacionar un objeto determinado. El control de las capacidades se realiza de forma anónima a la identidad del poseedor de la capacidad, lo cual resulta muy útil en sistemas distribuidos y con granularidad fina, donde el volumen de objetos es muy grande.

A diferencia de las listas de control de acceso, el mecanismo de capacidades respeta por tanto la independencia entre objetos que se establece como requisito de un sistema integral orientado a objetos. El control de acceso a un objeto (servidor) es totalmente independiente de los objetos (cliente) que pretendan utilizar el objeto dado. Con listas de control de acceso, el objeto servidor debe conocer todos los objetos cliente que lo puedan utilizar.

9.2.5 Integración en el modelo de objetos

Los objetos se comunican a través de mensajes. La invocación de un método de un objeto implica el envío de un mensaje a este objeto. Antes de que el método se ejecute, el sistema habrá de comprobar que está permitida esta invocación al objeto cliente. La capacidad que debe enviar el cliente puede ser incluida como parte integrante del mensaje que llegue al servidor. El mecanismo de protección actuará justamente antes de la invocación del método, comprobando los permisos que hay en la capacidad.

El uso y comprobación de las capacidades se realiza de forma paralela al propio mecanismo de invocación de métodos que posee el modelo de objetos. Este mecanismo por tanto no se ve prácticamente afectado por la introducción de protección, sino que sirve de estructura para integrar ésta, sin necesitarse estructuras de datos adicionales que haya que incluir dentro de los propios objetos o asociados a éstos.

El mecanismo de protección, por tanto, se integra dentro del modelo de objetos de una manera fluida, sin necesitar cambios adicionales en el mismo, más que añadir la propia comprobación de permisos de acceso.

[1] Que en cualquier caso habría que utilizar, aún usando listas de control de acceso (éstas la usarían para la parte del objeto servidor).

9.2.6 Flexibilidad

Las capacidades son entidades que se utilizan en el nivel de objetos, y no requieren la introducción de conceptos primitivos adicionales en un sistema de objetos, como el concepto de usuario. El concepto de usuario debe ser totalmente independiente del sistema básico de protección para que éste pueda cumplir el requisito de flexibilidad deseado. La aparición de usuarios en relación con la seguridad puede realizarse siempre a un nivel superior, dentro de la política de seguridad y no en el mecanismo básico de protección.

Así por ejemplo, en un sistema que use capacidades como mecanismo básico de protección se puede implementar una política de seguridad por encima que asigne las capacidades en función de los usuarios. Se podría establecer un servicio de denominación de objetos (o en estructura más clásica un sistema de ficheros) donde cada objeto cuente con una serie de capacidades, por ejemplo para el propietario, para el grupo y para el resto, simulando así la estructura del UNIX. En función del usuario que pida la capacidad, se le devolverá una u otra.

No obstante, con este tipo de control de acceso básico, también se pueden implementar políticas muy diferentes (no necesariamente ligadas a la existencia de usuarios), constituyendo un sistema más flexible.

9.3 Análisis del modelo mixto

El modelo mixto, visto en el capítulo 4 es un modelo basado en capacidades, en las que no aparecen los permisos directamente, sino que aparece una clave. En el lado del servidor el mecanismo de protección dispondrá de una lista de control de acceso en la que cada entrada tiene dos campos: clave y permisos. En función de la clave se comprueban los permisos y se concede o no el acceso.

Este mecanismo tiene la ventaja de proteger las capacidades contra posibles falsificaciones por parte de clientes maliciosos, pero también tiene el inconveniente de la recarga que significa la búsqueda en la lista de control de acceso por cada petición.

El modelo mixto permite la revocación selectiva, que puede ser individualizada, si cada clave se corresponde con un cliente o en grupo, si varios clientes con los mismos permisos poseen la misma clave. En este caso la eliminación de una entrada en la lista de control de acceso supone la eliminación de los permisos a un conjunto de clientes.

Este modelo aunque posee más ventajas que las listas de control de acceso posee los mismos inconvenientes que estas: la sobrecarga de la búsqueda en las tablas, la necesidad de conocer el conjunto de objetos del sistema y la difícil escalabilidad por el tamaño y cantidad de tablas.

9.4 Análisis del modelo de retículo de seguridad

En este modelo se asigna a los sujetos una acreditación o categoría y se asigna a los objetos una clasificación o tipo. Por otro lado existen unas reglas que rigen la clasificación. En función de estas reglas se obtendrán diferentes modelos.

Se trata de un mecanismo de acceso no discrecional, pues existe una política claramente establecida que indica cuáles son los posibles accesos de un sujeto cualquiera, sin que éstos puedan ser modificados discrecionalmente (a voluntad del propietario del objeto).

El hecho de tratarse de un mecanismo de acceso discrecional restringe su uso al ámbito especializado de aplicación para el que se creó inicialmente. Para un sistema integral orientado a

objetos, donde existen miles de objetos, cada uno de los cuales con un nutrido conjunto de operaciones a las que se quiere restringir el control de manera individualizada es una opción demasiado especializada. Un sistema como este no encaja con los requisitos planteados fundamentalmente por la falta de flexibilidad que impide la aplicación del mecanismo de seguridad a diferentes ámbitos.

9.5 Introspección en pila

Este mecanismo se desarrolló para la plataforma Java, y consiste en buscar información acerca de la procedencia de la clase que realiza la invocación en la pila de llamadas que contiene información de las sucesivas invocaciones a métodos de las diferentes clases. En función de la procedencia se buscan los permisos de que dispone la clase (en esencia en una lista de control de acceso) para decidir si se puede llevar a cabo la operación deseada o no.

Este mecanismo, al realizar búsquedas en listas sufre de los mismos problemas básicos de las listas de control de acceso. Por otro lado, necesita un conocimiento de la estructura interna del sistema (en este caso de la máquina virtual de Java) y una manera de acceder a esas estructuras internas para obtener la información necesaria. En general la implementación de este mecanismo no sería portable ya que estas estructuras internas podrían variar en diferentes implementaciones de la máquina de Java. Este problema se agrava si se intenta aplicar a otros sistemas, como el sistema integral orientado a objetos propuesto, en el que la máquina abstracta no especifica ninguna estructura interna determinada para su implementación (por ejemplo una pila de llamadas), para favorecer las posibilidades de implementación de la máquina.

Aunque desarrollado para la plataforma Java, se han propuesto variantes avanzadas de este mecanismo para proteger el código móvil en Java e incluso se propone una variante (estilo de paso seguro) como complemento de seguridad para un sistema basado en capacidades [Wal99].

9.6 Comportamiento de los modelos respecto a los problemas de seguridad más comunes de un sistema

En este apartado se describe el comportamiento de las listas de control de acceso y las capacidades frente a los principales problemas de seguridad, descritos ya en el capítulo 3.

9.6.1 El problema de la propagación (conspiradores en comunicación)

El modo de funcionamiento de las capacidades permite que un objeto cualquiera envíe una capacidad que posea a otro objeto. De este modo le está pasando los permisos de acceso sobre un tercer objeto. Por supuesto, siempre y cuando exista la posibilidad de comunicación entre estos dos objetos, y además ambos objetos deben estar de acuerdo (compinchados). Es lo que se denominaba el problema de los conspiradores en comunicación. En ocasiones esto no es deseable, puesto que lo que se quiere es dar capacidad a un objeto específico (autorizado) y sólo a ese, sin que exista la posibilidad de que otros objetos (cómplice) accedan al objeto (confidencial). Es decir, que el objeto autorizado no pueda propagar el acceso a objetos cómplice no autorizados.

En un sistema puro de capacidades, en el momento en que se pasa una capacidad a un objeto, ya no se tiene ningún tipo de control sobre si éste envía la capacidad a otros objetos. Esto resulta en que un objeto autorizado podría pasar la capacidad de acceso a un objeto confidencial a cuantos objetos cómplice desease (siempre que tuviese un canal de

comunicación con estos objetos, es decir, que el objeto autorizado tuviese a su vez capacidades de acceso a estos objetos cómplice).

Lo que en principio parece un inconveniente derivado del uso de capacidades, en realidad es un problema de difícil solución, puesto que otros sistemas, como las listas de control de acceso, que aparentemente no presentan este problema, en realidad también lo sufren.

9.6.1.1 Propagación indirecta: la ilusión de la solución del problema de la propagación con listas de control de acceso

Así pues, veamos, por ejemplo, el caso de las listas de control de acceso, en las que, aparentemente, este problema no ocurre. Para que un objeto (autorizado) tenga permisos sobre otro (confidencial) necesita estar incluido en su lista de control de acceso, y eso sólo lo puede hacer el servicio de protección (por encargo del propietario) y no un objeto (cómplice) cualquiera por su cuenta. Sin embargo, siempre es posible que un objeto cómplice tenga acceso al objeto confidencial, utilizando al objeto autorizado como intermediario. Si un usuario *UAutorizado*, posee permisos sobre un objeto *AConfidencial* y se desea que otro usuario *UCómplice* también los posea, puede conseguirse simplemente haciendo que *UCómplice* llame a *UAutorizado* cada vez que quiera algo sobre el objeto *AConfidencial*. *UAutorizado* simplemente tiene que replicar la interfaz del objeto *AConfidencial*, dar acceso a esa interfaz a *UCómplice* y hacer de pasarela entre estas llamadas y las llamadas reales al objeto *AConfidencial*.

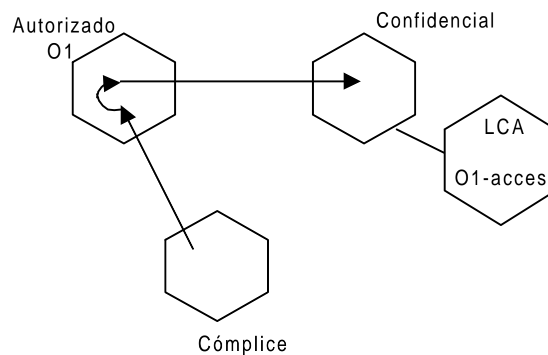


Figura 9.8: Propagación indirecta en mecanismos con LCAs

Este es un modo de acceso indirecto al objeto *AConfidencial* pero el resultado es el mismo que si el usuario *UCómplice* tuviera directamente los mismos permisos que *UAutorizado*. El resultado es en definitiva el mismo que en el caso de las capacidades: en esencia una vez que se autoriza el acceso a un *UAutorizado*, en realidad se está autorizando a ese usuario y a todos aquellos a los que este decida pasar los permisos (bien de manera directa si es un sistema de capacidades puro, bien de manera indirecta si es mediante listas de control de acceso).

Haciendo una analogía con la vida real, en el caso de las capacidades funcionan como si para entrar a un edificio se diera la llave de éste a las personas autorizadas. Aquél que posea la llave podrá entrar sin ningún tipo de problema. Y además, el poseedor de una llave siempre puede realizar copias de la misma y dárselas a otras personas, propagando la entrada a cuantos él quiera. En el caso de las listas de control de acceso, no se da la llave, sino que se pone a un portero en la puerta, el cual posee una lista de los clientes que pueden entrar. Evidentemente un cliente no tiene acceso a la lista para modificarla añadiendo un amigo a ésta. Sin embargo, si desea que el amigo obtenga la información que hay en el edificio, siempre se puede prestar él mismo como intermediario, de forma que cuando el amigo desee algo del edificio se lo

comunica al cliente registrado, éste entra sin problema, toma la información y a continuación se la envía a su amigo.

9.6.1.2 Expresión de la prohibición de propagación

En el caso de las capacidades no se puede expresar dentro del modelo esta prohibición de propagación. Por tanto el modelo no puede asegurarla. En el caso de las listas de control de acceso si se puede expresar (teóricamente) esta prohibición, simplemente no incluyendo al objeto cómplice en la lista de control de acceso del objeto confidencial. A pesar de ello, el sistema no puede cumplir con esa prohibición debido a que es posible utilizar intermediarios.

Por tanto, queda demostrado que el problema de la propagación no es un problema exclusivo de las capacidades, sino que existe en ambos mecanismos de protección. Es pues un elemento intrínseco a la seguridad, tanto en sistemas informáticos como en la vida real. En caso de las listas de control de acceso, el problema incluso se agrava, pues pueden dar la sensación errónea de que evitan este problema (pueden expresar la prohibición), y hacer que el usuario confíe en que la propagación está totalmente controlada, cuando en realidad no es así (no pueden asegurar el cumplimiento de esa prohibición).

Como nota adicional, existe la posibilidad en un sistema de capacidades extendido, de evitar que las capacidades puedan ser redistribuidas libremente como en un sistema puro. Se trata de añadir un permiso adicional sobre la propia capacidad que impida la libre copia o redistribución de la misma. Sin embargo, esto sigue sin evitar el problema de la utilización de intermediarios.

9.6.1.3 Irresolubilidad del problema de la propagación

Como se ve, el problema de la propagación, en el que los conspiradores se pueden comunicar, no tiene solución. La solución pasa por evitar que exista esa comunicación, con lo que el problema se transforma en el problema del confinamiento, que se trata más adelante. Como se verá, con el paradigma de listas de control de acceso, no hay solución satisfactoria al mismo.

9.6.2 El problema de la revocación

El problema de la revocación es quizás el mayor problema con que cuenta el mecanismo de capacidades puro. Dado que no existe ningún registro de qué objetos poseen capacidades, una vez que se concede una capacidad no es posible la eliminación de ésta. Es decir, no se puede revocar el acceso a un objeto, una vez concedida la capacidad, puesto que se desconoce cuantas copias de la capacidad existen, y dónde se encuentran.

9.6.2.1 La ilusión de la revocación selectiva con listas de control de acceso

En el caso de la utilización de listas de control de acceso, en principio no ocurre esto, pues para eliminar permisos sobre un objeto lo que se hace es eliminar la entrada de la lista de control de acceso de dicho objeto correspondiente al cliente que se desea revocar. Sin embargo, esto sólo es efectivo si la consulta de la tabla se hace en cada invocación de la operación. Ya se ha visto en capítulos anteriores que los sistemas como Guide, o Choices, que utilizan listas de control de acceso, restringen el número de comprobaciones de la lista a una, siendo ésta la primera vez que se desea realizar la operación. A partir de ahí se establece un enlace que permanece durante toda la ejecución (un modo de uso al estilo de las capacidades, en realidad), por razones de eficiencia. Esto significa que tampoco se pueden revocar permisos una vez concedidos la primera vez a través de la lista de control de acceso.

9.6.2.2 Revocación no selectiva de capacidades

En realidad, un sistema puro de capacidades no puede hacer una revocación selectiva de acceso: revocar el acceso a un objeto determinado al que previamente se le concedió. Sin embargo se puede hacer fácilmente una revocación total, modificando el identificador del objeto al que se quiera revocar totalmente el acceso. Al modificar el identificador quedan invalidadas automáticamente todas las capacidades que hubiera repartidas por el sistema sobre dicho objeto, pues dentro de la capacidad, además de los permisos se almacena una referencia al objeto (a través de su identificador). Al cambiar el identificador todas las capacidades apuntarían a un objeto inexistente, consiguiéndose una revocación total. Para recuperar las capacidades selectivamente habría que volver a enviarlas una a una a todos los objetos excepto a los seleccionados que se quisieran revocar.

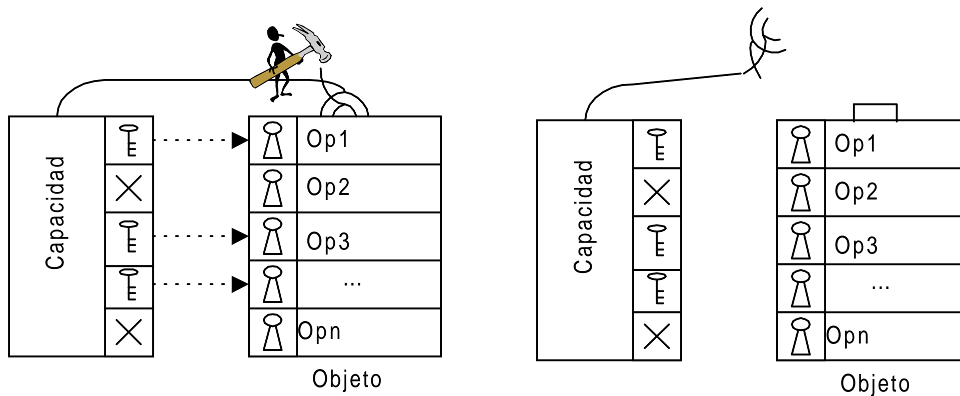


Figura 9.8: Revocación total con capacidades

No obstante, los sistemas que actualmente permiten de una u otra forma revocación de permisos, como es el caso de Grasshopper, que utiliza un modelo mixto de capacidades y listas de control de acceso, la protección está limitada a objetos de granularidad muy gruesa, (contenedores y loci) que restringe considerablemente el número de controles de protección. En sistemas integrales orientados a objetos, con objetos de granularidad muy fina, no sería factible la aplicación de este método.

9.6.2.3 Revocación selectiva con capacidades: fachadas

En un sistema puro de capacidades es posible construir mecanismos de revocación selectiva, lo que da prueba de la flexibilidad de un sistema puro de capacidades. Se trata de determinar sobre qué objetos se necesita revocación selectiva. Para dar acceso a estos objetos, en lugar de hacerlo directamente se crea un objeto “fachada”, que reproduce la interfaz del objeto y que es el que tiene acceso directo al mismo. Al resto de los objetos se les concede acceso a este objeto en lugar de hacerlo directamente al objeto “verdadero”. La revocación simplemente consiste en eliminar la referencia de la fachada al objeto original, haciéndola no válida. Se consigue una revocación selectiva creando una “fachada” individual para cada objeto o grupo de objetos a los que se quiera aplicar la revocación selectiva.

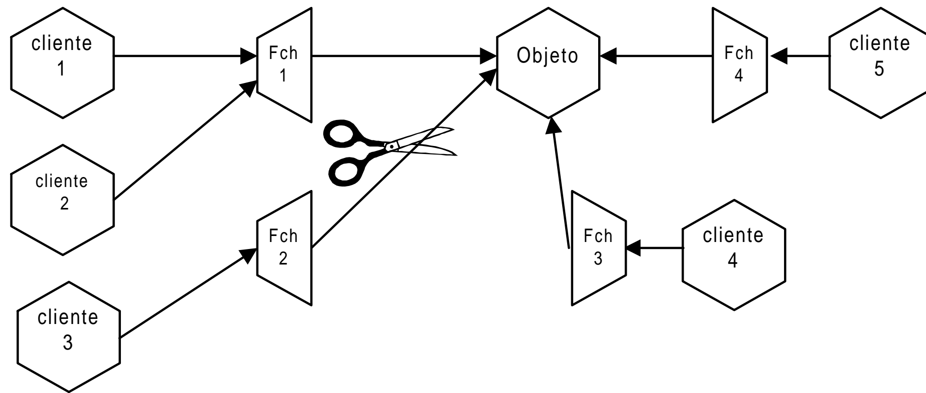


Figura 9.9: Uso de fachadas para permitir la revocación selectiva de permisos en sistemas con capacidades.

Este patrón de uso (similar al patrón de diseño "fachada") puede extenderse haciendo que la fachada sea más inteligente. Por ejemplo, podrían crearse fachadas de un sólo uso, que una vez utilizadas la primera vez se auto-revocaran. Pueden desarrollarse otros patrones de diseño para otros casos específicos, como por ejemplo capacidades de uso exclusivo [MSC97] que aseguran que el objeto que las posea tiene la exclusividad en el acceso a un determinado objeto.

Este es un buen ejemplo de la flexibilidad que ofrece un sistema de capacidades puro. El mecanismo básico puede extenderse con construcciones de alto nivel como las fachadas para conseguir un mecanismo con revocación selectiva. Sin embargo, en la mayoría de las situaciones en las que no es necesaria revocación selectiva, el mecanismo básico es suficiente y no se necesita utilizar estas construcciones adicionales, ahorrando sobrecarga y código adicional.

9.6.2.4 Extensión del monitor de referencia para la revocación selectiva

En aquellos sistemas en los que fuera necesario implementar la revocación selectiva de manera estandarizada, puede utilizarse un monitor de referencia. Este monitor intervendría en el funcionamiento de las capacidades de manera transparente, y sería el encargado de devolver "fachadas" en lugar de capacidades normales para aquellos objetos importantes que necesitasen revocación selectiva. Así mismo sería el encargado de realizar físicamente la revocación selectiva. El funcionamiento normal del mecanismo de protección ya es en sí un monitor de referencia. Se trata de extender este funcionamiento normal para conseguir otras propiedades. En un sistema integral orientado a objetos puede lograrse esto con flexibilidad utilizando la reflectividad como mecanismo de extensión [Álv98], en este caso para extender el funcionamiento del mecanismo de protección. Otras propiedades que se podrían implementar en el monitor, además de la revocación selectiva pueden ser la trazabilidad (determinar qué objetos utilizan las capacidades), estadísticas de utilización de objetos a través de las capacidades (para equilibrado de carga) o el confinamiento, como se verá más adelante.

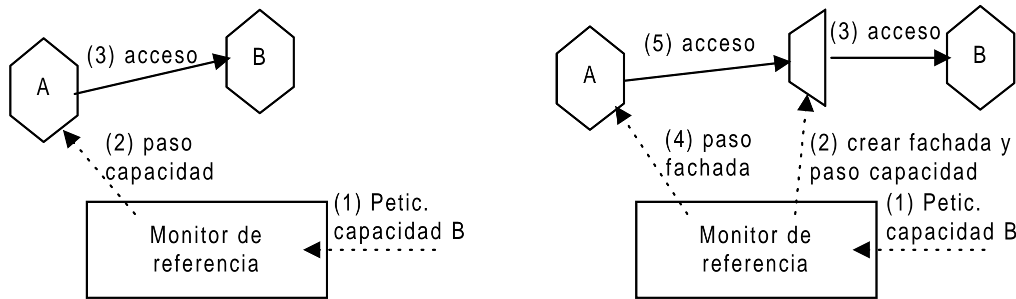


Figura 9.10: Uso de un monitor referencia, que da una capacidad directamente a un objeto normal y una fachada para un objeto importante

9.6.3 El problema de la delegación (representante confuso)

Los sistemas que utilizan listas de control de acceso, en los que el dominio de protección está asociado a un usuario, se encuentran en ocasiones con la necesidad de extender los permisos de un usuario temporalmente. Esto es necesario para realizar ciertas operaciones para las cuales el usuario directamente no tiene permiso, pero que necesita cuando está ejecutándose un determinado proceso. Un caso arquetípico es el del juego de Kowalski, descrito en el capítulo 3, en el que el usuario que ejecuta el juego necesita actualizar la lista de puntuaciones global, aunque en principio no debe tener acceso libre a la lista. Un caso similar es el de la actualización del fichero de claves en un sistema Unix.

Los sistemas basados en listas de control de acceso suelen introducir mecanismos análogos al bit *setuid* del Unix para intentar resolver este problema. Sin embargo, debido a la propia concepción de las listas de control de acceso, esto producía una serie de problemas adicionales (ver capitulo 9)

Este problema no se produce con el uso de capacidades. En un sistema de capacidades el dominio (la autoridad) va asociado al proceso u objeto. El acceso al fichero de puntuaciones correría a cargo del juego, y no del usuario que está jugando en un momento dado. Bastaría con que el juego cuente con una capacidad de acceso a las puntuaciones globales con los permisos adecuados para solucionar el problema.

El usuario puede ejecutar el juego normalmente y es el propio juego el que accede a las puntuaciones globales sin problemas, puesto que el juego se ejecuta no con la autoridad del usuario inicial (listas de control de acceso), si no con su propia autoridad, por la que puede acceder a las puntuaciones. Esto no necesita la introducción de mecanismos específicos (parches) para resolver (parcialmente) el problema de la delegación, como en las LCA.

En términos del problema del representante confuso (descrito en el capítulo 3), también se resuelve con el uso de capacidades. El usuario del compilador no deberá enviar al compilador el nombre del fichero al cual quiere enviar la información de depuración. En su lugar, deberá enviarle una capacidad más limitada, la capacidad más restringida que necesite el compilador. En este caso será una capacidad para escribir en el fichero de estadísticas. Esta capacidad debe estar en manos del usuario para poder enviarla al compilador. Lógicamente, el usuario nunca podrá disponer de una capacidad de este tipo para el fichero de Facturas, que el administrador habrá pasado previamente al compilador, por lo que no existe riesgo de violación de seguridad (que el usuario pueda "engañar" al compilador para sobreescribir el fichero de facturación).

9.6.4 El problema del confinamiento

Como se explica en el capítulo 3 de "Conceptos de Seguridad", se trata de asegurar que los datos que se entregan a una aplicación sólo se utilizan para los fines exactos que manifiesta la especificación de la aplicación, y quedan confinados a los límites de la misma. Es decir, si un programa recibe unos datos para procesarlos de cierta manera, no debe permitirse que este programa utilice estos datos con otros fines que no sean los acordados, de manera que pueda guardar información o pasarla a terceros para luego utilizarla en beneficio propio. Es decir, un programa servidor al que se ceden unos datos (o una autoridad) para hacer un determinado trabajo no debe poder transmitirlos a un tercer programa (espía), aunque tenga la intención de hacerlo (esté compinchado).

El problema del confinamiento es un problema de alto nivel. Hay que asegurarse de alguna manera la confidencialidad de los datos. Tanto las capacidades como las listas de control de acceso son mecanismos estrictamente de protección, y no están pensados para resolver directamente este tipo de problemas.

Para su resolución habrá que asegurarse de que el programa no puede guardar los datos de forma privada una vez que los ha recuperado, ni de transmitirlos a otros procesos. Es decir, debe garantizarse que no exista un canal de comunicación entre el servidor y el espía, que haría que se estuviese en el caso de los conspiradores en comunicación, que no tiene solución. En un sistema con LCA esto no es posible dentro de un control de acceso discrecional, que es el que se necesita un sistema integral orientado a objetos de propósito general.

Para lograr el confinamiento con listas de control de acceso, se necesita garantizar que el servidor no pueda añadir al compinche a su propia lista de control de acceso, para poderse comunicar con él. Es decir, no se permite cualquier modificación por parte del propietario de una lista de control de acceso. Sin embargo, esto supone un sistema de control de acceso no discrecional, y se necesita un sistema de control discrecional en el que cada sujeto decida quién puede acceder y quién no al propio objeto, es decir, que el propietario de un objeto puede modificar libremente su lista de control de acceso.

En el caso de que el compinche estuviera en la lista de control de acceso del servidor, no existe una manera sencilla y transparente que evite que se comuniquen entre sí.

9.6.4.1 Confinamiento con capacidades: monitor de referencia

En el caso de las capacidades, esto se transformaría en comprobar que nunca existe una capacidad que permita comunicar el servidor con el compinche. Sin embargo, esto no cambia el sistema de funcionamiento discrecional de las capacidades, que consiste en que cada objeto decide a voluntad a qué otros objetos le pasa las capacidades, pero **por canales (capacidades) previamente existentes**. Simplemente comprobando que estos canales no existen se logra el confinamiento.

Por otro lado, aunque existiera un canal de comunicación entre ambos objetos (en principio abierto), existe una manera de cortar la comunicación antes de que se produzca. Para ello habría que controlar las operaciones de paso de parámetros (capacidades) para verificar que no se consiga pasar información no autorizada de acuerdo con lo que se necesite confinar. De nuevo esto puede realizarse haciendo que en el sistema exista un monitor de referencia que controle de manera transparente el funcionamiento de las capacidades en el sistema: además del control de acceso normal comprueba que no se violan las propiedades del confinamiento. En un sistema integral orientado a objetos, este monitor puede construirse si fuera necesario utilizando la reflectividad y manteniendo la flexibilidad del sistema.

En cualquier caso, puesto que el decidir qué es lo que se debe confinar es ya parte del terreno de una política, más que del control de acceso no se tratará con más profundidad, aunque existen trabajos que demuestran que pueden construirse sistemas basados en capacidades que tienen la propiedad del confinamiento.

Como nota adicional, hasta el momento sólo existe una demostración formal de que un mecanismo de acceso verdaderamente consigue el confinamiento, y esta sólo es para los sistemas basados en capacidades [SW97].

9.7 Elección de Capacidades como mecanismo de protección

El mecanismo de protección que se pretende desarrollar deberá ser adecuado para un sistema integral de objetos en los que éstos se encuentran distribuidos. Los objetos son entidades autónomas, que pueden migrar de un nodo a otro. Para facilitar la migración en un solo paso, los objetos son entidades activas autocontenidas, encapsulando no sólo los datos del objeto sino también la información semántica relevante para el objeto (hilos, información de protección, etc.). De aquí la importancia de que el estado del objeto contenga la menor carga posible.

Teniendo esto en cuenta, un mecanismo de protección basado en Listas de Control de Accesos no parece muy adecuado. Las LCA requieren una lista por cada objeto del sistema con los objetos que pueden acceder a ella. Todos los objetos deben guardar y ser conocedores de todos los objetos que puedan usarlos. Existe por tanto una dependencia muy grande entre objetos que no es en absoluto recomendable en un sistema integral. Por otro lado, estas listas crecen rápidamente, y su número sería muy elevado teniendo en cuenta que el sistema integral soporta objetos distribuidos, y pretende realizar protección al nivel de objetos de cualquier granularidad, tratándolos a todos de igual manera.

Por otra parte, este tipo de protección introduce una complejidad adicional en el sistema, forzando el uso de identificación del objeto y sistema de registro para cada objeto, necesaria para construir la lista de control de acceso.

El modelo mixto cuenta con los mismos inconvenientes de las listas de control de acceso, por lo que también se descarta; y los modelos de flujo de información resultan demasiado inflexibles al obligar a un tipo de acceso no discrecional.

El uso de capacidades como mecanismo de protección, aunque cuenta con algunos inconvenientes, como la revocación, o la posible sobrecarga excesiva, tiene sin embargo, considerables ventajas sobre el resto de los mecanismos.

9.7.1 Integración en el modelo de objetos como característica destacable

El análisis realizado ha conducido a la consideración de que las capacidades constituyen el mecanismo de protección que mejor encaja con un modelo de objetos uniforme. La inclusión de un mecanismo de capacidades en un sistema de objetos autosuficientes permite que los objetos se mantengan como entidades completamente autónomas, puesto que el mecanismo no introduce relaciones de dependencia con otros objetos.

Para el acceso a un objeto hace falta la capacidad, al igual que hace falta la referencia. Como las referencias a un objeto, una capacidad será obtenida creando un nuevo objeto o mediante el paso de parámetro en una invocación de un método. Una vez obtenida, la capacidad se envía con el resto de los parámetros del método.

El funcionamiento de las capacidades sigue, por tanto, un mecanismo paralelo al funcionamiento de las referencias a los objetos. Como se verá en el próximo capítulo 11 de diseño del modelo de protección, en un SIOO las capacidades se integran dentro del modelo de objetos. La integración de capacidades y referencias en una nueva entidad serán la clave de todo el mecanismo.

10 Estudio de Capacidades

Las capacidades están cobrando especial importancia como mecanismo de protección en los nuevos sistemas de computación. La utilización masiva de la programación orientada a objetos y la necesidad de protección entre objetos que se comunican entre sí, ha propiciado la utilización de capacidades por ser este mecanismo uno de los que mejor se adaptan a las nuevas técnicas.

Aunque la idea básica permanece la misma, han sido utilizadas una amplia variedad de técnicas en la implementación y administración de capacidades. A continuación, se van a detallar las principales características generales del uso de capacidades como mecanismo de protección.

Las capacidades fueron propuestas inicialmente por Dennis y Van Horn como una técnica para describir la semántica del acceso controlado a los datos. Aunque el término capacidad fue introducido por Dennis y Van Horn [DH66], el concepto tiene su origen en las *codewords* de Iliffe y Jodeit [IJ62]. Por otra parte, el trabajo de Lampson [Lam71] establece una relación formal entre listas de control de acceso y capacidades.

10.1 Conceptos generales sobre capacidades

10.1.1 Qué es una capacidad

Básicamente puede considerarse que una capacidad es un descriptor que apunta a un objeto e incluye información de protección para dicho objeto. Se trata, pues, de una referencia a un objeto junto con un conjunto de permisos sobre las operaciones de ese objeto.

Una capacidad consiste en:

- Identificador único para un objeto, que permita localizar el objeto.
- Conjunto de permisos para el objeto. Pueden ser sencillos (rwx) o con más semántica (operaciones de alto nivel sobre un objeto).
- Información de estado opcional para la propia capacidad. Por ejemplo, operaciones que se pueden realizar sobre ella, como copiar, eliminar, restringir, etc.

Identificador	Permisos	Estado

Figura 10.1: Elementos de una capacidad

Elementos generales de una capacidad

En el caso más simple, una referencia a un objeto puede ser vista como una capacidad para un objeto. En este caso existirían, por omisión, permisos de acceso a todos los métodos visibles por el usuario. En el caso general, se incluirían permisos para todas las posibles operaciones que tuviera el objeto.

10.1.2 Modo de operación

La posesión de una capacidad se asemeja a la posesión de una llave, que permitirá abrir una o varias puertas. El mero hecho de poseerla dará acceso al local, independientemente de quién sea el que la posea. La cerradura constituye la parte de seguridad con que cuenta el local para protegerse de intrusos. De esta manera, las capacidades, permiten el acceso a los recursos y son la única manera de acceder a los mismos[1].

La posesión de una capacidad es condición necesaria y suficiente para tener acceso al objeto al que se refiere

Cuando se desea acceder a un objeto, el mecanismo de protección comprueba la existencia de la capacidad con los permisos adecuados (si hay un permiso para la operación que se intenta invocar) y en función de ello admite o deniega el acceso.

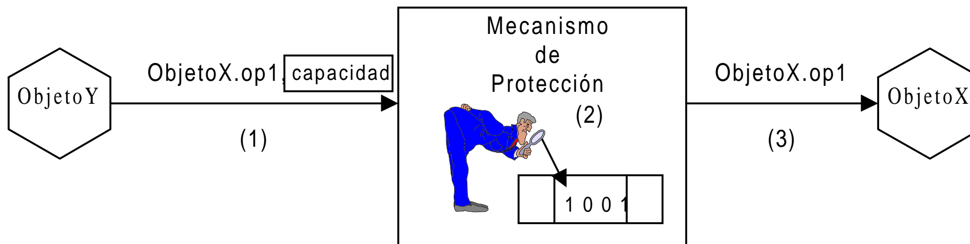


Figura 10.2: Modo de operación de las capacidades

10.1.2.1 Expresividad

Un concepto clave, que va unido al concepto de capacidad, es el de "expresividad" [MSC97]. No es posible decir a alguien que haga algo si no se tiene forma de expresar la petición. Si un objeto no puede expresar una petición, no hará falta comprobar si se permite que haga o no la petición. Un ejemplo de ello puede verse en las interfaces gráficas, donde sólo aquellas opciones que tiene sentido usar están activas y pueden ser usadas. Para aquellas opciones que no pueden usarse, ni siquiera puede intentarse su utilización.

De esta manera, la seguridad mediante capacidades toma frecuentemente una forma mediante la cual nunca se da a un objeto no fiable la posibilidad de expresar una petición peligrosa. Es decir, nunca se le da una capacidad a ese objeto no fiable que le permita expresar esa petición. Este concepto está relacionado con el principio de mínimo privilegio.

10.1.3 Obtención de capacidades

Existen dos formas [MSC97] de que un objeto *B* pueda adquirir una capacidad para un objeto *C*:

- Crear el objeto *C*, con lo cual se devolverá una capacidad de acceso al mismo en su creación.
- Recibir una referencia a *C* (una capacidad) a través de un tercer objeto *A* que la poseyera previamente (siendo pasada como parámetro de una operación de *B* invocada por *A*).

  Para que *A* envíe a *B* una referencia a *C*, *A* debe tener una referencia a *C*, y una referencia a *B*.

[1] Al menos si se desea asegurar la protección de todos los recursos del sistema.

Puesto que *A* tiene la elección de dar a *B* la capacidad para *C* o no dársela, estamos ante una situación discrecional. Puesto que *A* debe tener una referencia a *B* para darle la capacidad, esta es una situación obligatoria.

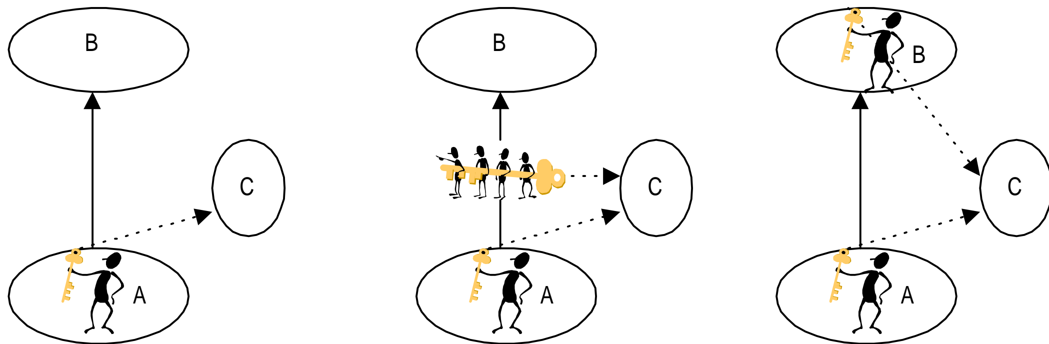


Figura 10.3: Recepción de una capacidad por parte de un objeto

10.1.4 Protección de las capacidades

Dado que la posesión de una capacidad otorga el derecho de acceso al objeto es necesario tener especial cuidado en las posibles manipulaciones que puedan hacerse de las capacidades.

La simple posesión de una capacidad es condición **necesaria y suficiente** para tener acceso al recurso. Por tanto, resulta imprescindible asegurar las capacidades ante cualquier posible tipo de falsificación o manipulación.

10.1.4.1 Falsificación de una capacidad

El tipo de implementación que se utilice para las capacidades, debe tener por tanto en cuenta el hecho de que el usuario no pueda de ninguna manera falsificar nuevas capacidades o manipular las ya existentes incorporando permisos que no tenía para ganar accesos no autorizados. Se expondrán, a continuación cuáles son los mecanismos de implementación más utilizados.

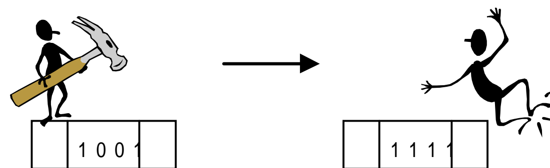


Figura 10.4: Falsificación de capacidades

10.2 Implementación de Capacidades

Existen varios planteamientos básicos de implementación de capacidades, que resuelven de diferente manera el problema de la falsificación y manipulación de las mismas. Se agruparán los diferentes mecanismos en tres grandes bloques [VVG95]:

10.2.1 Etiquetado hardware

En este diseño de arquitectura, cada palabra de memoria tiene un bit extra (etiqueta). Estos bits no son accesibles directamente por procesos de usuario y son utilizados como marcas que indican la contención o no de una capacidad. La colocación y la lectura de esta etiqueta sólo pueden ser realizadas mediante instrucciones en modo supervisor. De esta manera el hardware restringe la manipulación de capacidades a rutinas del sistema operativo. Así, el usuario no podrá falsificar ni manipular capacidades.

Este método tiene el inconveniente de que es preciso utilizar un hardware específico, con lo que se condiciona totalmente su uso, y no resulta por ello una buena elección actualmente. No obstante existen ejemplos de sistemas que utilizan este tipo de implementación, como Monads [RA90].

10.2.2 Segregación

Las capacidades se almacenan en un área de memoria protegida (por ejemplo por segmentación), que no puede ser manipulada por los procesos de usuario. En este caso, no es necesario un hardware específico, sino que el sistema operativo mantiene las capacidades separadas en una zona de memoria sólo accesible por él. Cualquier operación que requiera capacidades debe pasar por el núcleo del sistema operativo.

Ejemplos de este tipo son KeyKOS [Lan89], Hydra [WLH81], o Grasshopper [DBF+94].

10.2.3 Capacidades dispersas

La capacidad es un conjunto de bits tomado de un espacio disperso muy grande. De esta manera la probabilidad de falsificar una capacidad es muy pequeña puesto que sólo son capacidades válidas un número reducido de las posibles. Esto permite que las capacidades se puedan almacenar de manera normal en el espacio del usuario. Se pueden usar dos variantes:

- **Claves (passwords)**. Se añade una clave a los campos de la capacidad. Para que se permita el acceso, esta clave tiene que coincidir con la clave almacenada en el objeto al que se accede. Esta clave debe ser imposible de generar por otros objetos que quieran falsificar la capacidad. Para ello se utilizan claves tomadas de espacios dispersos (muy grandes) para que la probabilidad de acertar la clave aleatoriamente sea muy pequeña. Ejemplos de capacidades *password* son Amoeba [TMV86], Mungi [VRH93] y Opal [CLF+93] entre otros.
- **Encriptación**. La propia capacidad contiene cifrados los permisos para que no puedan ser manipulados.

Núm. Bits 48	24	8	48
Puerto Servidor	Objeto	Permisos	Campo de comprobación

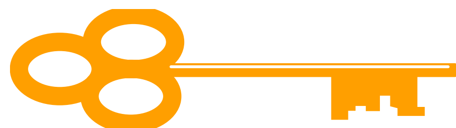


Figura 10.5 : Capacidad en Amoeba

La ventaja de las dos primeras soluciones, el etiquetado y la segregación es que es imposible que el usuario pueda manipular o crear capacidades falsas puesto que es el sistema el único que puede acceder a la información de una capacidad. Sin embargo esto requiere un hardware especial en el primer caso y supone la existencia de espacios de memoria "tradicionales" en el caso de la segregación.

La ventaja de las capacidades dispersas (solución "software") es que permiten su almacenamiento dentro de las propias estructuras de usuario, como "punteros" a objetos, aunque por ello requieren un control adicional para evitar su falsificación.

10.3 Permisos sobre los métodos que portan las capacidades

Las capacidades transportan los permisos que el poseedor de la capacidad tiene sobre el elemento a proteger, es decir, sobre el recurso. En los sistemas existentes basados en capacidades se encuentran diferentes alternativas en cuanto a cuántas operaciones de los objetos se protegen: todo/nada o bien un número fijo de permisos.

10.3.1 Todo/Nada

La utilización de este tipo de capacidades significa que la posesión de la capacidad sobre un objeto implica la posibilidad de acceso a cualquier operación de éste y si no se posee no se puede acceder en absoluto al mismo. La granularidad de protección es al nivel de objetos en su conjunto, y no al nivel de métodos individuales. Un ejemplo de este tipo de sistemas es Choices [CIM+93].

En los sistemas con lenguajes de programación orientados a objetos puros, una función análoga la realiza la referencia al objeto. Efectivamente, en estos sistemas, para poder acceder a un objeto es necesaria su referencia, y con ella es posible acceder a todas sus operaciones. Si no se tiene una referencia no se puede acceder al objeto.

10.3.2 Número fijo de permisos

En este caso se establece un número fijo de permisos determinado por el sistema, que determina el número de operaciones que se pueden proteger de un objeto.

El tipo de permisos que se almacenan depende del recurso a proteger. En la mayoría de los sistemas que utilizan este mecanismo de protección, las capacidades se utilizan para acceso a recursos muy concretos como puede ser la memoria (segmento, bloque, página, etc.) o los ficheros. En este caso, los permisos que se introducen son tan simples como lectura, escritura, y en algunas ocasiones, ejecución. De esta manera todos los recursos comparten las mismas operaciones, y por tanto siempre tienen los mismos permisos.

Un ejemplo de sistema existente que integra capacidades de este estilo, que restringen el número de permisos que pueden utilizarse es Mungi [CIM+93]. En Mungi todos los objetos tienen las mismas operaciones, y por tanto el número de permisos es fijo e igual para todos los objetos. Esto es debido a que únicamente ofrece como objetos los segmentos y las páginas, relacionados con la gestión de una memoria virtual global persistente, y los permisos se reducen a lectura, escritura, ejecución y destrucción.

En otros casos, como en el propio sistema Unix, aunque éste utiliza listas de control de acceso, los permisos se reducen a lectura, escritura y ejecución en los ficheros, que son los únicos que considera para su protección.

Otros sistemas, como Amoeba [MRT+90] o KeyKOS [RHB+86], que permiten el uso de objetos definidos por el usuario dentro de procesos servidores, con métodos arbitrarios, establecen un máximo de 8 bits dentro de la capacidad reservados para un máximo de 8 permisos relativos a métodos de un objeto. Por tanto no se pueden proteger más de 8 métodos de un objeto, lo que parece restrictivo.

10.4 Permisos relativos a la propia capacidad

Además de los permisos relativos a las operaciones que ofrecen los objetos, es posible introducir permisos adicionales, relativos a operaciones de manejo de la propia capacidad, para así restringir su manipulación.

10.4.1.1 Permiso de copia de la capacidad

Permite efectuar copias de la capacidad para posteriormente enviarla a otro objeto, por ejemplo.

El uso de este permiso depende del tipo de capacidades que se implementen. En capacidades segregadas o etiquetadas será necesaria esta operación si se desea poder hacer copias, puesto que las capacidades no pueden manipularse directamente. Habrá una operación del sistema operativo que permita realizar una copia de las mismas, la cual puede ser denegada si el permiso de copia sobre la capacidad no está activo.

En sistemas donde utilizan capacidades dispersas, que están bajo el control del usuario, éste siempre puede realizar copias sobre ellas al estar almacenadas en el espacio de usuario, por lo que en principio este permiso no tendría demasiado sentido.

10.4.1.2 Permiso de eliminación de instancias

Este permiso hace referencia a la posibilidad de destruir un objeto a través de una capacidad que apunta al mismo. En la mayoría de los casos, la simple posesión de una capacidad de acceso a un objeto no debe ser suficiente para poder eliminar el objeto, si el sistema dispone de primitivas destinadas a esta función. En estos casos este permiso indicaría si el usuario puede o no realizar esta función a través de la capacidad.

En algunos sistemas no existe posibilidad de eliminar directamente un objeto y sólo existe la recolección de basura. En estos casos este permiso no tendría sentido. En sistemas en los que puedan convivir ambas modalidades sólo afectaría este permiso a la utilización de la eliminación directa.

10.4.1.3 Permiso de eliminación de la propia capacidad

En este caso se trata de controlar la eliminación de la propia capacidad, no de la eliminación del objeto al que apunta la capacidad. En algunas situaciones puede ser necesario impedir que un usuario pueda eliminar una capacidad determinada, por ejemplo una capacidad que da acceso a un recurso del sistema necesario para la ejecución de las aplicaciones del usuario.

Evidentemente, la eliminación de la capacidad implica que el objeto que la posee deja de tener acceso al objeto al que se refiere la capacidad, a no ser que tenga otras capacidades sobre el mismo objeto.

10.5 Operaciones intrínsecas a las capacidades

10.5.1.1 Restricción de permisos

La eliminación de ciertos permisos dentro de una capacidad es una operación deseable en un sistema con protección. Si se posee una capacidad y se desea enviar a otro objeto, podrá enviarse con los mismos permisos, o en ocasiones se deseará enviar pero con permisos aún más restringidos que todos los que se poseen. Así por ejemplo, cuando se crea un objeto generalmente el creador recibe una capacidad con permisos para todas las operaciones del objeto. Si éste desea enviar la capacidad a otro objeto restringirá previamente los permisos dejando únicamente los estrictamente necesarios para así cumplir el principio de mínimo privilegio. En muchos casos se especifican los permisos que se quieren activar mediante una máscara en la que previamente se indican qué métodos se quieren dejar activos. Evidentemente, sólo se podrá dejar activo un método si la capacidad inicial lo tenía activo (sólo se puede restringir permisos).

Parece lógico que esta operación esté siempre disponible y por tanto se puedan restringir permisos siempre que se desee. De lo contrario lo único que se conseguiría sería el tener que enviar capacidades a otros objetos con más privilegios de los necesarios, poniendo en peligro la seguridad del sistema.

Este principio de mínimo privilegio también implica que no pueda existir una operación para ampliar los permisos existentes en una capacidad. Si así fuera, cualquier objeto podría adquirir fácilmente más permisos de los que se le habían concedido inicialmente, rompiendo este principio.

10.5.1.2 Duplicación (copia) de capacidades

La realización de una copia de la capacidad es una operación posible, que consiste en obtener un duplicado de la propia capacidad. Esta operación puede estar protegida con el permiso de copia visto anteriormente. Es una operación necesaria si se quiere enviar capacidades con diferentes permisos a diferentes objetos. Para ello se duplica la capacidad inicial y se aplica la operación de restricción de permisos de manera particularizada a cada copia.

En muchos casos la operación de copia y de restricción de permisos se combinan en una operación única en la que además de copiar la capacidad se restringen a la vez los permisos. En muchos casos se utiliza una máscara definida previamente por el usuario para indicar qué permisos se quieren dejar activos en la capacidad resultante. Como es lógico, y para seguir el principio del mínimo privilegio, en la capacidad resultante sólo puede quedar activo un permiso si ya estaba activo en la capacidad original.

10.5.1.3 Creación de capacidades

Otra operación que se necesita es la que permite la creación de las propias capacidades, que permite crear una nueva capacidad, antes de empezar a utilizarla. Normalmente se proporciona mediante una llamada al sistema en sistemas con capacidades segregadas.

10.5.1.4 Inicialización de capacidades

Una vez creada una capacidad se plantea la necesidad de cómo hacer que esta tome un valor (se inicialice) para apuntar a un objeto determinado. El mecanismo de funcionamiento de las capacidades indica que esto sólo puede realizarse de dos maneras: mediante la creación de un nuevo objeto y la devolución de la capacidad inicial de acceso al mismo (en la que

normalmente estarán activos todos los permisos), o mediante un paso de parámetros que dé valor a la capacidad.

10.5.1.5 Revocación de capacidades

Las características intrínsecas de las capacidades puras, impiden la posibilidad de revocación, puesto que, una vez que se da una capacidad a un cliente no se deja constancia de quién la ha recibido, ni se puede controlar si éste la transmite a terceras personas.

Sin embargo, algunos sistemas amplían el mecanismo de capacidades para controlar la revocación, añadiendo un permiso a las capacidades que indica si está revocada o no, junto con una operación correspondiente. En estos sistemas es necesario añadir un complejo entorno adicional para conseguir esto que complica grandemente el sistema, cuando la mayoría de las aplicaciones no lo requieren o bien puede lograrse por otros métodos ya mencionados, como fachadas de nivel superior, que proporcionan un nivel de indirección que permite implementar la revocación.

10.6 La tecnología de orientación a objetos y las capacidades

En sistemas desarrollados utilizando tecnología de orientación a objetos, en los cuales los objetos deben comunicarse entre sí, el uso de capacidades resulta especialmente adecuado. De manera resumida, pueden esbozarse las razones que hacen que la protección basada en capacidades puede tener una buena integración con las tecnologías orientadas a objetos. La llamada a un método de un objeto supone el envío de un mensaje a éste. Dentro del propio mensaje puede añadirse la capacidad que da permiso para el acceso al método. El objeto llamado, o un servicio de seguridad que se interponga, comprobará si efectivamente existe el permiso para el método, en cuyo caso se dará paso a la operación.

10.7 Implementación del mecanismo de comprobación de permisos

La localización del mecanismo de protección de permisos es otro aspecto que puede variar en un sistema con capacidades. De alguna manera el sistema tiene que comprobar la información de protección presente en una capacidad cuando se pretende invocar una operación de un objeto a través de ella. Si existe el permiso adecuado para la operación en cuestión, se deja proceder la invocación, en caso contrario se devuelve un error al objeto que origina la invocación (mediante una excepción, normalmente). A continuación se van a analizar las diversas posibilidades: integración en el núcleo del sistema operativo, en servidores del usuario, en tiempo de compilación y en el soporte en tiempo de ejecución del lenguaje

10.7.1 Integración el núcleo del sistema operativo

El hecho de que el mecanismo de protección se encuentre localizado en el núcleo del sistema operativo es la solución más habitual que suele encontrarse en la mayoría de los sistemas, tanto si están basados en capacidades como si no lo están.

Normalmente el mecanismo no es único y uniforme, sino que depende del recurso a proteger, por lo que el sistema cuenta con varios algoritmos que controlan la seguridad. Algunos sistemas operativos modernos amplían el ámbito de protección de recursos proporcionando también protección para servicios al nivel de usuario, con lo que se complica por tanto, el mecanismo. En cualquier caso, es el sistema operativo el que comprueba que el

acceso esté permitido, sobre la base de capacidades, listas de control de acceso, o cualquier otro medio que se emplee para ello.

Amén de los sistemas operativos clásicos, y comerciales, que tienen una estructura monolítica, como UNIX, sistemas operativos como KeyKOS centralizan todo el sistema de protección en el propio micronúcleo del sistema operativo. KeyKOS utiliza capacidades segregadas cuya gestión es llevada a cabo por el micronúcleo, quien se encarga de comprobar la existencia o no de permisos en la capacidad cuando se intenta acceder a un objeto.

10.7.2 Integración en servidores del usuario

Otra posibilidad que establecen algunos sistemas es la de incluir toda o parte de la seguridad en servidores de usuario.

Esto ocurre por ejemplo en sistemas como Amoeba, donde el núcleo sólo establece el mecanismo básico de paso de mensajes entre cliente y servidor y comprueba que la dirección del servidor sea correcta. Sin embargo, el mecanismo que comprueba la existencia o no de permisos en la capacidad que envía el cliente forma parte del objeto servidor, que debe encargarse de implementar la comprobación de capacidades de manera correcta.

Los servidores que ofrece Amoeba son procesos a nivel del usuario que gestionan un conjunto de objetos del mismo tipo. Se puede hablar, en este caso, de que el mecanismo de protección se implementa al nivel de usuario y se apoya en el micronúcleo que proporciona Amoeba para su funcionamiento. Cualquier usuario que implemente un nuevo servicio deberá incluir su propia implementación del mecanismo de protección si desea restringir el acceso a los recursos de su servicio.

No obstante, el propio micronúcleo utiliza el mismo sistema internamente para aquellos servicios que se encuentran dentro de él, obteniendo así un modo de protección bastante uniforme.

Un caso similar ocurre con el sistema Opal, en el que los procesos se ejecutan en dominios de protección y cuando se llama a un segmento de memoria situado en otro dominio se utilizan los portales que identifican al servidor que maneja el segmento. La comprobación de permisos se establece a nivel del servidor y la de identificación del servidor a nivel del núcleo.

10.7.2.1 Inconvenientes la integración en servidores de usuario

El uso de este modelo implica que sea el propio usuario programador de aplicaciones el que se encargue de programar o incluir el núcleo de seguridad junto con el proceso servidor cada vez que implemente un nuevo servicio.

Falta de uniformidad

El mecanismo no es uniforme al estar su implementación en manos del programador de aplicaciones. Cada servidor puede tener características propias en cuanto a la implementación del mecanismo de comprobación de permisos.

Dificultad para granularidad fina

Este mecanismo está pensado para proteger objetos de grano grueso, que son los que ofrecen los servicios implementados. Una protección al nivel de objetos de grano fino aportaría una sobrecarga excesiva, puesto que para cada invocación de un objeto, aún siendo éste muy pequeño, debería activarse la rutina de comprobación de permisos, rutina que, recordemos, se encuentra en cada servidor de usuario.

Dificultad de movilidad

Los objetos estarán asociados a procesos servidores que se encargan de su ejecución. El mecanismo de protección va ligado al servidor, por lo que los objetos no podrán moverse del lugar donde se encuentre el servidor. Lo que sí puede hacerse es mover el servidor completo.

Falta de Mediación total

El principio básico de diseño denominado "Mediación total", afirma que en un sistema no debe poderse esquivar el mecanismo de seguridad. Dado que es el usuario el encargado de establecer la protección, puede decidir no aplicar este mecanismo, incumpliendo este principio.

Mayor posibilidad de agujeros de seguridad

Otro problema parecido se presenta cuando el usuario implementa incorrectamente la comprobación. Esto es aún peor puesto que el usuario piensa que está seguro, cuando en realidad no lo está.

10.7.3 Integración al nivel de compilador

El sistema operativo Choices realiza la comprobación de permisos en tiempo de compilación cuando un objeto intenta acceder a otro objeto de otro espacio de direcciones. En este caso comprueba la lista de control de acceso, y si los permisos son adecuados crea un objeto representante (*proxy*) que se encarga de hacer de enlace entre los espacios de direcciones en tiempo de ejecución y se enlaza en el módulo ejecutable en tiempo de compilación.

10.7.4 Integración al nivel del soporte en tiempo de ejecución del lenguaje

Con la aparición de los *applets* de Java, la plataforma incorpora un mecanismo de protección frente a accesos no autorizados al sistema de ficheros local, que se suministra mediante un conjunto de paquetes que forman parte del soporte en tiempo de ejecución (biblioteca estándar) del lenguaje y la plataforma Java. Este mecanismo va evolucionando en las sucesivas versiones que aparecen y, como se explica en el capítulo 8, cuando un *applet* intenta realizar una operación sobre discos locales, se pone en marcha el mecanismo de protección para comprobar si dicho *applet* tiene permiso para realizarla. Asimismo, navegadores web, como Netscape o HotJava utilizan mecanismos similares en su soporte en tiempo de ejecución (*runtime*).

Así por ejemplo, se establece protección en aquellas llamadas que registran peligro, como las de lectura y escritura en disco. Estas funciones de biblioteca se modifican para que previamente a la realización de la operación se llame al soporte en tiempo de ejecución para que se compruebe el tipo de cargador que se ha utilizado para cargar la clase que pide la operación, y en función de éste se localizan sus permisos en una lista de control de acceso y se permite o deniega el acceso.

10.8 Ventajas de las capacidades en general

10.8.1 Doble función de las capacidades: Denominación y protección

Además de servir como mecanismo de protección, las capacidades ofrecen la posibilidad de actuar como mecanismo de denominación. Entre la información que contienen debe existir un identificador único para el objeto, este hecho permite que la presentación de una capacidad para acceder a un objeto no suponga una sobrecarga adicional, sino que va unida a la necesidad de enviar el nombre del objeto.

10.8.2 Independencia cliente/servidor

Obsérvese que el uso de capacidades mantiene independientes a los clientes de un recurso respecto al servidor que soporta ese recurso. Es decir, el servidor no necesita conocer ninguna información respecto al cliente, la mera presentación de la capacidad es suficiente para que éste pueda conceder o denegar el acceso. Esto es importante en sistemas distribuidos en los que el número potencial de objetos que acceden a un servidor puede ser muy elevado.

10.8.2.1 Eficiencia

La validez de un acceso es fácilmente comprobable, simplemente se trata de examinar directamente si se tiene el permiso adecuado o no. La no necesidad de comprobación de quién es el cliente simplifica considerablemente la operación. Un acceso por parte de un cliente es implícitamente válido si se posee la capacidad. Dado que si no se tiene una capacidad no se puede intentar siquiera el acceso, el número de operaciones que se deben comprobar se reduce enormemente frente a otros mecanismos que permiten intentar cualquier acceso.

10.8.2.2 Simplicidad

Es simple debido a la relación entre los permisos y la denominación. La capacidad simplemente añade permisos al mecanismo de denominación, sin recargar sustancialmente el mecanismo.

10.8.2.3 Flexibilidad

Es flexible porque un sistema de capacidades permite al usuario decidir sobre las capacidades definiendo patrones arbitrarios de autorización. Es decir, el usuario discrecionalmente decide a quién le cede las capacidades. Esto no impide que puedan utilizarse políticas de seguridad de tipo obligatorio, como se describe en el siguiente apartado.

10.8.2.4 Flexibilidad en el número de paradigmas diferentes que pueden ser implementados

Por otro lado, como mecanismo de protección de bajo nivel, puede ser utilizado por niveles superiores para implementar políticas de seguridad diferentes. Estos niveles superiores simplemente establecerían los permisos de acceso en las capacidades de acuerdo con lo que dicte cada política determinada.

10.8.2.5 Escalabilidad

Las capacidades resultan adecuadas para sistemas ampliables. El comportamiento es el mismo independientemente del número de objetos que deban ser protegidos, no siendo así en el caso de las listas de control de acceso, en cuya técnica, a medida que el sistema crece, las listas de control aumentan de tamaño y también la sobrecarga de protección.

10.9 Inconvenientes de las capacidades en general

10.9.1.1 Control de propagación

En un sistema puro de capacidades, cuando un sujeto pasa una copia de una capacidad para un objeto a otro sujeto, el segundo sujeto puede pasar copias de la capacidad a muchos otros sujetos, sin que el primero tenga conocimiento de ello. En algunas aplicaciones puede ser necesario el control de la propagación de capacidades.

La propagación de una capacidad puede ser controlada añadiendo un permiso adicional, descrito anteriormente, denominado "permiso de copia" de la capacidad. La propagación de la capacidad puede prevenirse desactivando este permiso cuando se proporciona una copia de la capacidad a otros usuarios, evitando así que pueda pasarse (copiarse) a otros sujetos. Otra forma de limitar la propagación es usar un contador de profundidad. El contador de profundidad va unido a cada capacidad, cuyo valor inicial es uno. Cada vez que se hace una copia de la capacidad, se aumenta en uno el contador de profundidad para la nueva copia. Existirá un límite en la profundidad. Cualquier intento de generar una copia de la capacidad cuyo contador de profundidad haya llegado al límite supondrá un error, limitando así, la longitud de la cadena de propagación de una capacidad.

Otra cuestión relacionada aparece en niveles más altos del sistema, en aplicaciones que necesitan un control de propagación más selectivo, dependiente de la política de seguridad. En estos casos no es el propietario de la capacidad el que decide si puede propagarse o no, si no que en función de la política de seguridad se determina a qué objetos puede propagarse una capacidad determinada y a cuáles no. Por ejemplo la política de seguridad puede determinar que un capitán no puede propagar una capacidad para un objeto "*top-secret*" a un objeto de rango inferior.

Este es un asunto relacionado con capas de más alto nivel del sistema y no con un mecanismo de protección de bajo nivel como son las capacidades, aunque pueden utilizarse como capa inferior para implementar políticas de este estilo. Como ya se apuntó en el capítulo 3, puede resolverse este problema implantando en el sistema un monitor de referencia que supervise las propagaciones de capacidades, comprobando que cumplen las restricciones impuestas por la política. Incluso podría utilizarse la información del permiso de copia, si existiera, como herramienta para representar estas restricciones al supervisar la propagación.

10.9.1.2 Revisión (trazabilidad)

Otro problema fundamental con las capacidades es la determinación de todos los sujetos que tienen permisos de acceso a un objeto. No existe pista alguna de quienes son los poseedores de las capacidades, con lo cual este problema es difícil de resolver en un sistema de capacidades puras.

El problema de la revisión es un problema relacionado con políticas de alto nivel en el sistema, más que con el mecanismo de control de acceso de bajo nivel del sistema. Este problema es similar al problema de la recolección de basura, de objetos a los que no apunta ninguna referencia. Se trata de la situación contraria, a partir de un objeto determinar las referencias que apuntan a él, y las posibles técnicas para determinarlas parecen similares a las usadas en recolección de basura. En un SIOO, donde las referencias están bajo el control de la máquina abstracta, la recolección de basura es fácil de implementar. De la misma manera podrían incorporarse algoritmos análogos para la revisión de capacidades.

La recolección de basura tiene un costo relativamente elevado, aunque los algoritmos están mejorando cada vez más hasta el punto de que ya se admite como una característica adecuada en los sistemas. En el caso de la revisión el problema está menos estudiado, aunque

es de suponer que se puedan realizar los mismos avances. Hay que tener en cuenta que la necesidad de revisión, al contrario que la recolección de basura, no afectará normalmente a todos los objetos, si no a aquellos que se consideren "importantes" y sea necesario conocer qué objetos pueden accederlos. Por tanto, el impacto en el sistema será menor, al necesitarse pocas revisiones.

10.9.1.3 Revocación de permisos de acceso

La revocación es un aspecto complicado debido a la posibilidad de propagación que tienen las capacidades. El sujeto que inicia la propagación pierde la pista de las capacidades que se conceden sobre dicho objeto y por tanto sería muy difícil revocarlas. La forma más simple de revocarlas es destruir el objeto o modificar su localización, con lo que todas las capacidades quedan anuladas. Ello implicaría tener que volver a enviar capacidades a aquellos sujetos que se desee. No sería, por tanto, una revocación selectiva.

Otro método es el que utiliza Amoeba. Cada objeto contiene un enorme número aleatorio, también presente en la capacidad. Cuando se presenta una capacidad para su uso, se comparan los dos números. Se permite la operación sólo en el caso de que estos números coincidan. El propietario de un objeto puede solicitar el cambio de este número, con lo cual se invalidan las capacidades existentes. No obstante, este esquema no permite la revocación selectiva; es decir, retirar el permiso a ciertos clientes pero mantenerlo para otros.

Otra forma de revocar las capacidades es haciendo que cada capacidad apunte a un objeto indirecto, en vez de apuntar al objeto en sí. Si el objeto indirecto apunta hacia el objeto real, el sistema siempre puede romper esa conexión, con lo que invalida las capacidades.

10.9.1.4 Recolección de basura

Cuando todas las capacidades sobre un objeto desaparecen del sistema, el objeto se vuelve inaccesible, convirtiéndose en basura. Si se desea evitar esto, se puede guardar información sobre el número de capacidades existentes en el sistema sobre cada objeto, y eliminar el objeto cuando éstas sean cero (recolección de basura).

11 Diseño del Modelo de Protección

En este capítulo se describen las decisiones adoptadas respecto al diseño del núcleo de seguridad ideado para formar parte de un Sistema Integral Orientado a Objetos, y que constituye el objeto de esta tesis doctoral. Esta descripción irá acompañada de elementos justificativos de las decisiones adoptadas.

Tres son los aspectos generales en los que se ha dividido el diseño del mecanismo y que son los siguientes:

1- Elección de capacidades como modelo de protección básico

2- Diseño de las capacidades

3- Diseño del mecanismo de comprobación de acceso

El capítulo 9 está íntegramente dedicado a justificar razonadamente cuáles son las causas que han llevado a elegir inicialmente las capacidades como el mecanismo más adecuado para implantar un Sistema Integral Orientado a Objetos.

Este capítulo, por tanto, se dedicará a describir y justificar cuál es el diseño específico elegido para la implantación de capacidades y el mecanismo de control de acceso propiamente dicho, aportando con ello una nueva concepción de núcleo de seguridad que proporciona beneficios con respecto a los implantados en los sistemas existentes.

11.1 Diseño de las Capacidades: Capacidades Orientadas a Objetos

En este apartado se van a justificar los aspectos relativos al tipo de capacidades seleccionado, en cuanto a la protección, y a los permisos específicos para capacidades que incluirán las propias capacidades.

11.1.1 Protección de las capacidades: Nuevo tipo de capacidades: Orientadas a Objetos

Como se ha descrito en el capítulo 10, los métodos tradicionales de implantación de capacidades para protegerlas frente a alteraciones se agrupan en capacidades hardware, segregadas y dispersas.

El etiquetado hardware hace que las capacidades sean manejadas únicamente por el hardware de manera controlada, marcando las posiciones de memoria ocupadas por las capacidades. Sin embargo, esta alternativa tiene el inconveniente de la necesidad de un hardware especial, lo cual resulta en estos momentos muy restrictivo. Teniendo en cuenta que un objetivo del SIOO es el de integrar grupos de máquinas heterogéneas mediante la implantación en las mismas de una máquina abstracta común, no puede restringirse las plataformas de ejecución del SIOO únicamente a las que tengan un hardware determinado. Esto descarta este tipo de capacidades.

Por otro lado, hoy en día las mejoras en el rendimiento de procesadores convencionales, debidas a las economías de escala, hacen menos posible la producción de hardware

especializado. Como consecuencia, este tipo de capacidades no es precisamente el más utilizado y se restringe únicamente a algún diseño realizado en décadas anteriores, en las que diseñar hardware especializado era económicamente más viable que en la actualidad.

Las capacidades segregadas están almacenadas en una zona de memoria de uso restringido al sistema, para evitar su falsificación. Esto impide que las capacidades puedan almacenarse en estructuras propias del usuario, lo cual sería más flexible. Además, el sistema operativo, se debe de encargar continuamente de su manipulación, ofreciendo al usuario una interfaz completa para su manejo (llamadas al sistema). Esto llevará consigo una sobrecarga de gestión que no aparecía por ejemplo en las etiquetadas hardware.

En un SIOO que persigue el objetivo de la homogeneidad y uniformidad, no pueden introducirse desequilibrios en el sistema, como los que impone el que los datos del usuario se almacenen en dos sitios distintos (espacio de usuario para los datos normales y espacio del sistema para las capacidades). Además, la propia gestión de las capacidades a través del SO introduce otro desequilibrio, al obligar a que éste proteja las capacidades de una manera diferente a la protección que se le ofrece al usuario para sus estructuras normales. En un SIOO, se pretende que todos los objetos sean homogéneos, y que no exista ningún tipo de objeto especial, ni siquiera los que formen parte del sistema operativo, que se traten de manera especial. Este mecanismo, por tanto, no es adecuado debido a estos problemas de homogeneidad y uniformidad.

Las capacidades dispersas permiten que el usuario pueda utilizarlas directamente, disponiendo de ellas en su espacio de direcciones y evitando así tener que realizar continuas llamadas al sistema operativo como ocurría en el caso de las segregadas. Para evitar la falsificación que pudiera producirse por parte del usuario se almacenan utilizando un conjunto de bits muy elevado de manera que la probabilidad de encontrar una secuencia que coincida con una capacidad es muy pequeña y además se encriptan las propias capacidades.

Este tipo de capacidades tiene la ventaja de que son fácilmente manejadas por el usuario, pero sin embargo producen una mayor sobrecarga, puesto que además de la sobrecarga introducida por el propio mecanismo de protección, será necesario comprobar también que la capacidad no ha sido falsificada, además de que cada capacidad ocupa más espacio del estrictamente necesario.

11.1.1.1 Capacidades Orientadas a Objetos

En esta tesis doctoral se propone utilizar para el SIOO un nuevo tipo de capacidades, que han sido denominadas "Capacidades Orientadas a Objetos" puesto 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 éstos. 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 siempre serán capacidades[1]. En este

[1] De hecho, en lo sucesivo y refiriéndose a este sistema, se utilizarán las denominaciones de referencia o capacidad indistintamente, ya que describen el mismo concepto.

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.

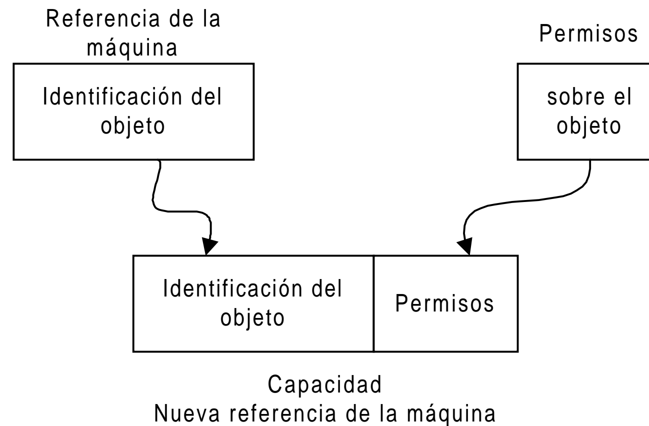


Figura 11.1: Nueva referencia de la máquina, equivalente a una capacidad

11.1.1.2 Referencias como capacidades básicas

Como es bien sabido, los lenguajes orientados a objetos puros basan su modo de trabajo en el uso y manipulación de las referencias que representan a los objetos. Cualquier invocación a un método de un objeto se hace a través de su referencia. Por tanto, la referencia es un modo de identificar al objeto de forma única.

Estos lenguajes encapsulan los objetos de forma que no es posible acceder directamente a los atributos de éstos si no es a través de algún método, y por tanto a través de su referencia. La referencia es por tanto el único modo de acceso a los objetos, siendo imposible cambiar directamente los mismos. En este sentido la referencia actúa como una capacidad. La posesión de la referencia es condición necesaria para el acceso a las operaciones del objeto, y por tanto a su información, y únicamente a través de las operaciones definidas al efecto.

Se puede decir, por tanto, que en los lenguajes de programación orientada a objetos de este estilo (no todos), que suelen denominarse “seguros”, las referencias actúan de capacidades para el acceso a los objetos. No obstante, estas capacidades son muy limitadas, puesto que no restringen el acceso de forma selectiva a las operaciones del objeto, únicamente dan la capacidad para el uso de cualquiera de sus métodos.

11.1.1.3 Ampliación de la información de las referencias

Partiendo de la premisa anterior, podría pensarse en ampliar la información existente en la referencia con información acerca de los permisos sobre los métodos.

En un SIOO las propias referencias pueden verse como objetos con sus atributos, que inicialmente se reducen al identificador del objeto al que apunta esta referencia. Si a esto le añadimos los permisos de acceso sobre las operaciones del objeto, la referencia se completa con información de protección, que será utilizada por el mecanismo de protección, y que convierte a ésta en una verdadera capacidad.

La conversión de las simples referencias en capacidades y el hecho de que éstas son imprescindibles para realizar cualquier operación sobre cualquier objeto, sea cual sea su granularidad, garantiza el uso de capacidades siempre y en cualquier nivel del sistema, e integra su uso en el modelo de objetos que soporta el sistema sin cambiar su manera de funcionamiento.

Dado que las capacidades (o simplemente referencias, a partir de este momento) son gestionadas únicamente por las instrucciones de la máquina definidas al efecto resulta imposible su falsificación, puesto que habría que acceder directamente a los atributos privados y la máquina garantiza que esto sea imposible. Esto elimina la necesidad de comprobar la posible falsificación de las mismas, y por tanto evita la sobrecarga correspondiente.

Por otra parte, el hecho de que las capacidades se encuentren fusionadas con las referencias permite su utilización libre como referencias normales en los objetos del usuario, lo que introduce mayor flexibilidad.

11.1.2 Permisos sobre los métodos que portan las capacidades: Número de permisos variable

11.1.2.1 Permisos sobre todos los métodos de un objeto: poco flexible

Como se ha explicado en el capítulo 10, un tipo de capacidades puede proteger el acceso al nivel de granularidad de un objeto, en cuyo caso, la posesión de la capacidad otorga permiso para acceder a cualquier operación del objeto.

Claramente, este tipo de capacidades resulta muy poco flexible. En la mayoría de los casos la protección se necesita al nivel de métodos individuales de un objeto (protección de granularidad fina). Por ejemplo, en un objeto "Calificación" de un estudiante, si se ofrece acceso (capacidad) al "Estudiante" para consultar su nota (método "consultar"), también se le daría acceso para cambiarla (método "cambiar"). Se necesita un tipo de capacidades más flexible que permita proteger individualmente los métodos de un objeto. Así al "Estudiante" se le podría dar acceso al método "consultar", pero no al método "cambiar". Con capacidades de este estilo, el otro tipo de capacidades es un caso particular de ésta, en la que todos los permisos están activos, y por tanto, se tiene acceso a todos los métodos del objeto.

11.1.2.2 Número fijo de permisos: demasiado restrictivo

Los requisitos de diseño impuestos para un Sistema Integral Orientado a Objetos como el elegido, donde la flexibilidad y la granularidad fina de protección se presentan como principales condiciones inclinan a decantarse por un sistema en el que la protección sea al nivel de métodos individuales de los objetos.

Dentro de las capacidades que permiten proteger métodos individuales de los objetos existen dos posibilidades: que el número de permisos que puede proteger una capacidad sea fijo (y por tanto limitado), o que éste sea variable (tantos permisos como métodos tenga el objeto).

La mayoría de los sistemas existentes que integran capacidades, restringen el número de permisos que pueden utilizarse a un número fijo determinado. Como ya se describió esto puede ser debido a la existencia de objetos de bajo nivel que tienen todos las mismas operaciones, como los ficheros de Unix (lectura, escritura, ejecución).

En otros casos la limitación a un número fijo de permisos es arbitraria, por ejemplo Amoeba restringe a 8 bits la información de protección. Por tanto, en cada objeto se pueden proteger únicamente 8 métodos (y en la práctica la mayoría de los objetos tienen más de 8). Si un objeto tiene más de 8 métodos no se podría ofrecer protección para todos los métodos. Esta limitación en el número de permisos puede venir determinada por necesidades de bajo nivel en la implementación del sistema, que hacen que se trate a las capacidades como registros de longitud fija para facilitar su implementación. Por otro lado, las aplicaciones de usuario "ven"

a las capacidades como tales registros de longitud fija y no como a una abstracción de alto nivel.

11.1.2.3 Capacidades Orientadas a Objetos: 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. Así pues, el uso de un número fijo de permisos no es adecuado, puesto que reduciría la protección a un número fijo de operaciones, dejando desprotegidas al resto de las implementadas en la clase.

Parece evidente que un mecanismo más flexible y completo consiste en un nuevo tipo de capacidades que de soporte a un número variable de permisos (tantos como métodos tenga el objeto). Éste modelo resulta aparentemente más complejo de solucionar y con mayor sobrecarga, puesto que el número de permisos depende del tipo del objeto al que hace referencia. Sin embargo, otros aspectos del diseño que se describirán posteriormente y los resultados de un prototipo contribuyen a la reducción o eliminación de los problemas que aparecen, teniendo en cuenta la conveniencia que aporta al usuario un sistema de protección de estas características.

Una ventaja del SIOO es que las referencias (ahora capacidades) 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 implantará, 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.

11.1.3 Permisos relativos a la propia capacidad: Salto de protección

En cuanto a los permisos que puede llevar una capacidad acerca de sí misma, como el de copia y el de eliminación, un criterio general dentro de los objetivos de un SIOO es el de economía de mecanismos y simplicidad. Esto hace que, como norma general, no se incluyan permisos o funcionalidad adicional de las capacidades si no es estrictamente necesario.

11.1.3.1 Permiso de copia: no necesario

Un permiso relativo a la propia capacidad que puede aparecer en las mismas es el permiso de copia, que permite o no obtener una copia de una capacidad existente.

En el SIOO existe una instrucción que permite duplicar (copiar) una referencia (la instrucción de asignación de referencias). Esta instrucción es básica en el lenguaje ensamblador de la máquina abstracta. En el diseño de capacidades introducido aquí (Capacidades Orientadas a Objeto), dado que la capacidad está integrada con la referencia, siempre podría realizarse una copia de ésta, mediante esta instrucción de asignación. La inclusión de un permiso de copia modificaría sustancialmente en principio el modelo de objetos establecido, ya que en algunos casos la asignación de referencias produciría una excepción de protección, lo cual no suelen esperar los programadores por lo general.

Por otra parte, el uso de este permiso no evita la posibilidad de que un objeto pase una capacidad a otro. Si un objeto tiene una capacidad y desea enviársela a otro puede hacerlo

independientemente de que haga una copia o no. Esto es posible simplemente pasándosela como parámetro de una invocación al objeto que debe recibirla. No resolvería nada el evitar que la capacidad se copie en este sentido.

Debido a estas razones se ha decidido no utilizar un permiso de copia de capacidades en el modelo diseñado.

11.1.3.2 Permiso de eliminación de instancias: protección de un método normal de todos los objetos

El permiso de eliminación permite o no eliminar la instancia a la que apunta una capacidad. Esto implica que el objeto que la posee deja de tener acceso al objeto al que se refiere la capacidad, a no ser que tenga otras capacidades sobre el mismo objeto, y que la capacidad propiamente dicha queda libre.

Esta eliminación del objeto es directa. El objeto se destruye incondicionalmente aunque haya otras referencias que apunten a él. El SIOO dispone de una instrucción que permite eliminar los objetos de esta manera. Dado que no es razonable que cualquiera que posea una capacidad que apunte a un objeto pueda eliminarlo, es conveniente que se proteja esta operación.

Sin embargo no es necesario incluir un permiso especial para esta operación, ya que esta operación puede verse como un método propio de la clase raíz de la jerarquía de clases del sistema. De esta manera, todos los objetos tendrían una operación de eliminación que podría invocarse a través de cualquier referencia. Simplemente protegiendo esta operación como un método más del objeto se consigue el mismo efecto que incluyendo el permiso de eliminación de instancias. Así pues, se logra el mismo objetivo de una manera más sencilla y uniforme, sin necesidad de añadir elementos adicionales al sistema. Este mismo mecanismo podría aplicarse a la operación de asignación de referencias si fuera conveniente, aunque, como se mencionó anteriormente, otras razones lo desaconsejan.

11.1.3.3 Permiso de eliminación de la propia capacidad: no necesario

Este permiso hace referencia a la posibilidad de destrucción de la propia capacidad (referencia). Cuando se destruye la referencia ya no se puede acceder al objeto que apunta (a menos que se posea por otro lado otra capacidad sobre el mismo). Sin embargo, el objeto al que apunta no se destruye. Una variante es en la que no se destruye la capacidad propiamente, si no que simplemente se asigna a otro objeto, o se libera (se hace que no apunte a ningún objeto, o al objeto nulo). El efecto es el mismo, salvo que la estructura de la capacidad sigue existiendo.

En el SIOO sólo existe la segunda variante, la destrucción de las referencias se realiza automáticamente cuando se abandona el método dentro del cual se definieron. Dado que la libre asignación de referencias es un recurso que se da prácticamente por supuesto en la programación orientada a objetos, no tiene sentido que se impongan restricciones en cuanto a la asignación de referencias por parte de los programadores.

En sistemas dotados de recolección de basura, la eliminación de la última referencia que apunta a un objeto provoca la eliminación del mismo. Más concretamente, en el momento en que a un objeto no apunte ninguna referencia (por destrucción de la misma o asignación a otro objeto), este objeto se destruye. La recolección de basura no afecta a la discusión de estos dos permisos de eliminación, si no que es ortogonal a la misma. La eliminación de un objeto al que no apunta ya ninguna referencia en el sistema puede realizarse siempre sin ningún problema y no afecta al mecanismo de protección en ningún aspecto.

11.1.3.4 Capacidades Orientadas a Objetos: Salto de la protección

Si la capacidad cuenta con un número variable de permisos, uno por cada método, el mecanismo de protección deberá localizar la posición que ocupa el permiso del método invocado en la capacidad. Esta operación podría implicar sobrecarga, teniendo en cuenta que un objeto puede tener métodos propios y otros heredados dentro de toda la jerarquía de clases.

"Salto de protección" para optimizar las llamadas a objetos no compartidos

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[1].

En los casos en los que la capacidad tiene todos los permisos activos, y por tanto el mecanismo de protección permitirá realizar cualquier operación, resultaría muy útil contar con un permiso adicional que indicaría "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. Evidentemente, si todos los permisos están activos, comprobar el permiso del método que se invoca es una pérdida de tiempo.

El mecanismo de protección puede empezar siempre mirando el estado del permiso correspondiente a "salto de protección". Si éste está activo, indica que todas las operaciones son permitidas, con lo que el mecanismo permite la invocación. En caso contrario, el mecanismo procede a la búsqueda del permiso de la operación invocada para comprobar su estado (comprobación normal de los permisos en la llamada a un método).

La primera comprobación añade una sobrecarga muy pequeña al sistema, comparado con la sobrecarga introducida en la búsqueda del permiso del método invocado. Teniendo en cuenta que en las aplicaciones orientadas a objetos la mayoría de las invocaciones a objetos se producen en objetos locales que no son compartidos, y normalmente no necesitan protección de sus métodos, el ahorro de carga es importante. Esto hace que por razones de eficiencia sea apropiado incluir este permiso en el diseño del sistema.

Todos los permisos para el creador

Puede verse este permiso como una suerte de indicador de propiedad, en el sentido de que el "propietario" de un objeto normalmente por omisión tiene todos los permisos sobre el objeto. Dado que en un SIOO inicialmente no existe la noción de propietario, esta analogía se puede establecer con el objeto creador de otro objeto. En estos casos lo adecuado es devolver una capacidad que tenga todos los permisos sobre el objeto que se crea, es decir, que tenga el permiso "salto de protección" activado.

Si después es necesario compartir el objeto, el creador puede duplicar la capacidad y aplicar operaciones específicas sobre la misma para indicar qué métodos serán aquellos que se pueden acceder a través de esa capacidad, como se verá posteriormente

El "salto de protección" visto como pista para la implementación

Si bien aquí se ha descrito un permiso "salto de protección", en realidad no es necesario que físicamente esta noción se implemente con un permiso diferenciado en las capacidades.

[1] Sin embargo, por razones de depuración, en algunos casos puede ser interesante mantener la protección en estos objetos para evitar, por ejemplo, llamadas a métodos a los que no tiene sentido acceder por la lógica de la aplicación.

Puede verse este “salto de protección”, como una abstracción para referirse al caso en que todos los permisos de una capacidad están activos. De esta manera la implementación subyacente puede utilizar esta abstracción para optimizar la implementación interna de la manera que considere adecuada.

Un ejemplo es la ya descrita optimización en la comprobación de permisos, motivación inicial de esta abstracción. Otra optimización puede ser de tipo espacial, puesto que en lugar de almacenar todos los permisos, cuando hay un “salto de protección” puede almacenarse la misma información de forma condensada, con un sólo bit, por ejemplo.

A pesar de esto, no es necesario que estas optimizaciones se lleven a cabo, puesto que el sistema funcionalmente se comportaría de la misma manera con ellas o sin ellas (únicamente menos eficientemente). Por tanto, este “salto de protección” puede verse como una especificación de diseño que da pistas a la implementación subyacente del mecanismo de protección para que sea más eficiente. El alto nivel del SIOO y de su máquina abstracta permite que estas optimizaciones de la implementación no afecten a las aplicaciones de usuario, al contrario que en otros sistemas (como por ejemplo los de capacidades de longitud fija).

Aunque no se conoce ningún sistema que utilice esta información, es precisamente este aspecto innovador el que permitirá que un mecanismo tan flexible y potente como el planteado, pueda llevarse a la práctica sin que suponga una sobrecarga infranqueable.

11.1.4 Operaciones intrínsecas a las capacidades

Debido al diseño introducido, en el que las capacidades forman parte de las referencias, automáticamente todas las operaciones que se aplicaban a las referencias en el SIOO ahora afectan a las capacidades (las nuevas referencias). Este es el caso de la copia de referencias (instrucción de asignación), la inicialización de referencia (instrucción de creación de instancias) y la creación de referencias. De esta manera se evita la introducción de mecanismos adicionales en el sistema, manteniendo la sencillez y homogeneidad del mismo, puesto que la semántica del modelo de objetos se mantiene totalmente. Desde el punto de vista externo, el funcionamiento del sistema no cambia.

La única funcionalidad adicional que introduce esta integración de las capacidades en el sistema es la necesidad de la operación de restricción de permisos, que es la única que es inevitable introducir para poder conseguir la implantación de un mecanismo de protección en el sistema.

11.1.4.1 Copia de capacidades: mecanismo ya existente

Esta copia la realiza la propia instrucción de asignación de referencias, que copia una referencia sobre otra (ahora copiará una capacidad sobre otra). Esta instrucción ya estaba presente en el sistema, por tanto no es necesario introducir una nueva instrucción específica en el sistema para realizar esta función.

11.1.4.2 Creación de las capacidades: mecanismo ya existente

La creación de las capacidades coincide con la anterior creación de referencias en el SIOO. Estas se crean al declarar los parámetros de un método o bien como variables locales de un método. De nuevo no es necesario introducir un mecanismo nuevo para las capacidades en el sistema, pues se aprovecha la semántica del mecanismo anterior para las referencias.

11.1.4.3 Inicialización de capacidades: mecanismo ya existente

De nuevo se corresponde con mecanismos ya existentes para el manejo de referencias. En la inicialización por paso de parámetros simplemente se copia el parámetro actual sobre el parámetro formal definido en el método. En el caso de la creación de una instancia a través de la capacidad, esta función está a cargo de la instrucción de creación de objetos ya existente.

En este último caso únicamente hay que hacer que la creación de objetos, además de devolver en la referencia el identificador del objeto, además coloque los permisos iniciales sobre el mismo. Como se describió en un apartado anterior, se ha escogido que los permisos iniciales estén todos activos, indicando que el creador de un objeto siempre tiene por omisión, e inicialmente, todos los permisos de acceso al mismo. Sin embargo, desde el punto de vista de las aplicaciones, la funcionalidad de la instrucción no cambia, estas adaptaciones son únicamente internas.

11.1.4.4 Restricción de capacidades: operación única “Restringir <capacidad>, <método>”

Hay un aspecto, sin embargo, que si debe tenerse en cuenta si se desea que desde un programa puedan ser manejadas las capacidades. Se trata de la posibilidad de que el usuario pueda restringir los permisos sobre métodos que posee una capacidad, antes de pasarla a otro objeto. Si un objeto posee una capacidad, podrá enviarla a otro, dejándola previamente con igual o menor número de permisos.

Esta es la única funcionalidad adicional que se necesita introducir en el sistema para implantar el mecanismo de protección. Puesto que es la única funcionalidad exclusiva y adicional, propia de un mecanismo de protección, es inevitable la inclusión de la operación de restricción de permisos.

Máscaras: no necesarias

Existen varias estrategias para introducir esta funcionalidad. Una de ellas es la utilización de máscaras de permisos, junto con operaciones específicas para la creación de estas máscaras.

Adicionalmente sería necesario incluir la propia operación de restricción de permisos utilizando estas máscaras. Otra alternativa sería la de modificar la instrucción de asignación de referencias añadiendo un parámetro adicional: la máscara de permisos a aplicar a la referencia asignada.

La estrategia basada en máscaras se desecha por su complejidad. Por una parte la propia existencia de máscaras necesita incluir operaciones adicionales para la creación de máscaras. Si además se modifica la operación de asignación de referencias para incluir máscaras, entonces se rompería la semántica existente del SIOO de manera innecesaria, siendo un objetivo el introducir el menor número de cambios necesario.

Instrucción única: “Restringir <capacidad>, <método>”

Siguiendo este principio de economía de mecanismo, se propone la inclusión de una única instrucción adicional: “Restringir <capacidad>, <método>”. Esta instrucción se limita a restringir en la capacidad el permiso de invocación del método indicado en el parámetro.

Con esta única instrucción se consigue toda la funcionalidad que permiten las otras estrategias de una manera más económica. Si es necesario restringir varios métodos simplemente se aplica de manera repetida esta instrucción, en lugar de aplicar la máscara. La creación de una máscara probablemente se tendría que realizar con tareas de este estilo, en cualquier caso, y sólo se sacaría utilidad si fuera necesario aplicar la misma máscara a un

número grande de referencias. Teniendo en cuenta que esto sólo sería necesario en un número ínfimo de casos y que de cualquier manera el compilador o un preprocesador podría encargarse de aplicar la instrucción repetidamente, la alternativa escogida parece más adecuada.

La implementación de esta funcionalidad no tiene por qué ser añadiendo una nueva instrucción al juego de instrucciones de la máquina. Una opción es implementarla como una operación más de la clase raíz de los objetos, de forma que todos los objetos tengan esta operación.

Esta será la única instrucción que se añade al modelo de objetos del sistema para poder manejar las capacidades. Este hecho es un elemento más que demuestran la simplicidad que proporciona la introducción de capacidades orientadas a objetos que se propone en esta tesis doctoral.

11.1.4.5 Revocación selectiva: no es necesario incluirla en el mecanismo básico

Los sistemas puros de capacidades no permiten la revocación selectiva, sólo una revocación total que puede lograrse cambiando el identificador del objeto e invalidando todas las capacidades que apunten a él.

Los sistemas que amplían las capacidades con previsiones para la revocación necesitan información adicional en las capacidades y complicados entornos para gestionar esta funcionalidad. Sin embargo, esto es poco flexible. Por una parte muchas aplicaciones no necesitan revocación en absoluto, y estarían pagando por una funcionalidad incluida en el sistema que no usan y que incluso reduce el rendimiento. Por otra parte, en caso de necesitarse, el número de objetos sobre los que se necesite revocación es muy reducido frente al número total de objetos y las mismas razones serían aplicables.

En estos casos, existen otros mecanismos más flexibles para proporcionar revocación que no necesitan introducirse en el mecanismo de protección básico del sistema, y sólo es necesario desplegarlos cuando la aplicación lo requiera. Un ejemplo que ya se mencionó en un capítulo anterior son las fachadas, que se implementan en el nivel de usuario y proporcionan un nivel de indirección que permite la revocación. Sin embargo, la sobrecarga necesaria para la revocación sólo se paga en los casos en los que se necesita revocación. Cuando no se necesita se sigue utilizando el mecanismo básico de capacidades, más eficiente.

11.2 Diseño del mecanismo de comprobación de permisos: implementación en la máquina abstracta

En cuanto al mecanismo de comprobación de permisos que examina los permisos de la capacidad cuando se realiza una invocación a un objeto para decidir si se permite el acceso o no, existen cuatro maneras principales de implementar: en el núcleo del sistema operativo, en servidores del usuario, en tiempo de compilación y en el soporte en tiempo de ejecución del lenguaje (véase el capítulo anterior).

En el caso de un SIOO, no existe un núcleo de un sistema operativo tradicional, con lo que no es aplicable este tipo de implementación. El sistema se compone de una máquina abstracta y de objetos normales de usuario, entre los cuales algunos tendrán funcionalidad asociada tradicionalmente al SO.

Se descarta también la implementación en los servidores de usuario (en el caso del SIOO sería en cada clase de objetos) por los inconvenientes descritos en el capítulo 10. Dejar la responsabilidad de la comprobación de la protección en manos de cada usuario va en contra

de muchos principios de diseño para el SIOO, fundamentalmente la violación del principio de mediación total, puesto que no se puede asegurar que todos los usuarios implementen el mecanismo de protección (y menos aún que lo hagan correctamente).

La implementación mediante bibliotecas del soporte en tiempo de ejecución de un lenguaje hace, por un lado, que el mecanismo de protección dependa de un lenguaje específico. Por otra parte, la protección se encontraría situada conceptualmente en el espacio del usuario, o por lo menos no formaría parte del núcleo fundamental del sistema. Esto da pie a una mayor facilidad para que intrusos pudieran saltarse el mecanismo de seguridad, bien encontrando fallos en la biblioteca, bien utilizando otros caminos que pudieran quedar abiertos en el sistema para acceder directamente al núcleo fundamental del mismo sin pasar por la capa externa del mecanismo de protección.

11.2.1 Capacidades Orientadas a Objetos: Implementación en la máquina abstracta

La existencia de una máquina abstracta orientada a objetos, que conforma el corazón del sistema integral, permite utilizar una vía diferente para implementar la comprobación de la protección. Se trata de implementarla como parte integrante de la propia máquina abstracta del sistema.

Dado que la máquina ya tiene integrado un mecanismo para realizar el paso de mensajes entre los objetos a través de referencias (que ahora serán capacidades), es sencillo introducir la implementación de la comprobación de permisos en el propio paso de mensajes. Así la máquina, cuando se envía un mensaje a un objeto, 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.

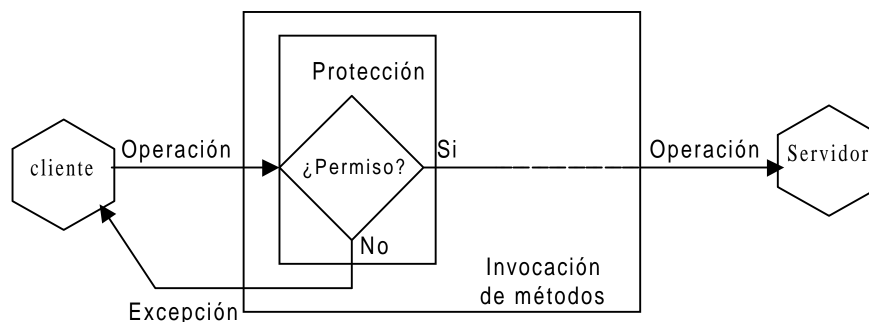


Figura 11.2: Integración del mecanismo de protección junto con la invocación de métodos, dentro de la máquina abstracta

Con esto se consigue mantener la uniformidad del modelo de objetos existente, puesto que externamente no se cambia la funcionalidad anterior del sistema (salvo por lo único necesario: la denegación de acceso cuando no se tienen permisos). 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, con lo que se garantiza la mediación total, y además es imposible saltarse el mecanismo ya que es parte del nivel más bajo del sistema.

11.3 Resumen de las características del modelo de protección

Es un modelo de protección uniforme, diseñado para un sistema integral orientado a objetos basado en una máquina abstracta, que se integra de manera fluida sin modificar el modelo de objetos. Sus características, expuestas resumidamente son:

1. Uso de capacidades como elemento base para la protección: control de acceso discrecional
2. Nuevo tipo de capacidades: Capacidades Orientadas a Objetos
   - Fusión de las capacidades y las referencias a objetos de la máquina
3. Diseño de las capacidades: Capacidades Orientadas a Objetos
   - Permisos al nivel de métodos individuales de los objetos
   - Número variable de permisos (tantos como métodos tenga el objeto)
   - Permisos relativos a la propia capacidad: Salto de protección (Todos los permisos activos, no se necesita comprobar)
   - Operaciones intrínsecas a las capacidades: Restricción de métodos (Restringir <método>, <capacidad>)
4. Diseño del mecanismo de comprobación de permisos: implementación en la máquina abstracta
   - Integración de la comprobación en el mecanismo de envío de mensajes de la máquina

11.4 Resumen del modo de operación del mecanismo

En los apartados anteriores se han analizado y aportado las decisiones fundamentales del diseño del mecanismo de protección, y se han dado las razones por las cuales se consideran las mejores opciones. En este apartado, se resumirá 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.

12 Ventajas del Modelo de Protección Diseñado

A continuación se van a describir las principales ventajas aportadas por este modelo, que corroboran la utilidad de éste:

12.1 Cumplimiento de los principios básicos de diseño de mecanismos de seguridad

El modelo de protección que ha sido desarrollado en esta tesis doctoral cumple los principios básicos de diseño exigibles a un sistema de seguridad [SS75]:

12.1.1 Mínimo privilegio

El mecanismo permite asignar permisos de manera individual para cada uno de los métodos de un objeto de cualquier granularidad (los permisos de las capacidades son de longitud variable y pueden establecerse con la operación de restricción) y para cada posible objeto cliente (a cada objeto se le pueden pasar las capacidades que necesite). De esta manera es posible conceder a cada objeto individual únicamente los permisos exactos que estrictamente necesite para trabajar con cualquier otro objeto, consiguiendo así el principio del mínimo privilegio. Al estar unidas las capacidades a las referencias, y ser estas la única manera de acceder a un objeto en la máquina, se asegura que no hay manera de evitar la exigencia anterior.

Así por ejemplo, si se desea ejecutar una aplicación externa, que debe utilizar recursos propios, como el disco, será importante que las capacidades enviadas a esta aplicación para que pueda acceder a los recursos sean los mínimos necesarios para realizar su trabajo sin que ello pueda suponer una amenaza contra elementos privados. El mecanismo de protección ofrece la posibilidad de restringir los permisos tanto como sea necesario para no favorecer los caballos de Troya.

12.1.2 Ahorro de mecanismos: robustez

Los mecanismos de protección complicados, que confían en la interacción de muchos elementos están expuestos a más agujeros de seguridad. En este caso, el mecanismo de protección es parte integral del sistema (integrado en las referencias y en el mecanismo de paso de mensajes de la máquina), y conceptualmente resulta muy simple. Esto hace que su implementación sea lo suficientemente pequeña y fácilmente verificable. Así la posibilidad de que se introduzcan agujeros de seguridad por errores en la implementación se minimizan.

Por otro lado, el mecanismo está localizado en la parte más interna del sistema, en lugar de colocado como un “parche” en capas superiores del sistema. Esto hace que sea virtualmente imposible esquivar el mecanismo de protección y acceder al núcleo del sistema por otro camino, pues el mecanismo forma parte del propio núcleo y no hay nada por debajo de él. Esto minimiza aún más los agujeros de seguridad, consiguiéndose un mecanismo de protección muy robusto.

12.1.3 Aceptación

Un buen mecanismo de seguridad debe ser fácil de utilizar para que éste sea aceptado por los usuarios y así cumplir su función. La fusión realizada entre referencias y capacidad hacen que el mecanismo sea conceptualmente muy simple, puesto que no añade nuevos elementos a las abstracciones del sistema que el usuario conoce. Todo funciona de igual manera, y prueba de ello es que cualquier aplicación antigua funcionará sin modificaciones con el nuevo mecanismo de protección integrado. Tan sólo existe una nueva operación que el usuario deberá aplicar si desea pasar capacidades con permisos restringidos. Por tanto, el mecanismo es francamente simple de usar y fácil de entender. Para un usuario que utilice el sistema no se añade complejidad adicional, con lo que no existirán problemas de aceptación del mecanismo.

12.1.4 Mediación total

Este principio implica la imposibilidad de esquivar el mecanismo de protección, sea cual sea la operación que se realice en el sistema. El mecanismo de protección ideado actúa en cada invocación de método, independientemente del método y del objeto que sea invocado. Esto asegura precisamente que la mediación sea total, asegurando que en ningún momento existe alguna parte del sistema que actúe de forma independiente a la protección. Todos los objetos, independientemente de que sean de un usuario normal, especial o del propio sistema operativo, se ven controlados siempre por el mecanismo de protección de forma obligada. Es decir, todas las operaciones del sistema están controladas, incluyendo el propio funcionamiento del sistema operativo. La inclusión del mecanismo en la máquina, junto con la invocación evita cualquier sobrepaso del mecanismo.

12.1.5 Diseño abierto

El mecanismo ideado, ha sido explicado aquí en su totalidad, y cualquiera puede ver su diseño. Se trata de un modelo no sujeto a secretos, que habitualmente acaban conociéndose y poniendo en peligro la seguridad del sistema. El diseño ideado no basa su bondad en la ocultación de información, sino en la implantación de éste en el nivel más interno del sistema, evitando cualquier tipo de manipulación por parte del usuario.

12.2 Cumplimiento de los requisitos de protección para un sistema integral orientado a objetos

12.2.1 Flexibilidad

La flexibilidad del mecanismo era uno de los requisitos que se propusieron para tener en cuenta a la hora de idear el mecanismo. Un mecanismo flexible será aquel que permite implementar distintas políticas de seguridad por encima de este, de forma que el sistema pueda adaptarse para su uso en entornos de computación diferentes.

Como se puede ver en el análisis realizado en el capítulo siguiente, el mecanismo de protección ideado es lo suficientemente simple y de bajo nivel, como para que puedan ser diseñados sobre él múltiples políticas. La inclusión de un permiso especial de Salto de Protección, que mejora el rendimiento, flexibiliza el mecanismo hasta tal punto que podría utilizarse en un sistema empotrado en el que no se desea ningún tipo de protección, sin perjudicar la eficiencia. El hecho de proteger los objetos de cualquier granularidad a nivel de todos sus métodos individualmente, y por tanto permite cumplir el principio de mínimo privilegio hace que no existan restricciones en cuanto a las imposiciones de permisos que pudiera establecer cualquier política que se implantara.

12.2.2 Movilidad

En un SIOO es deseable que los objetos que se encuentran en el sistema puedan cambiar de ubicación de manera transparente, incluyendo esto que la movilidad no suponga problemas de protección. Como se describe en el capítulo 6 sobre requisitos, el modelo de objetos ofrece un objeto como entidad autónoma que contiene su propia semántica completa, y por tanto no depende de otras entidades externas. Cuando un objeto se desplaza, lleva por tanto toda la información necesaria para continuar con su ejecución, entre ella las referencias a los objetos que necesita. La fusión de capacidades y referencias añade de forma simple la información de protección entre la información del objeto, sin tener que modificar en ningún momento los aspectos relacionados con la traslación de la información de un lugar a otro. La única diferencia será que ahora la cantidad de información de las referencias es algo mayor.

Puesto que la transparencia en el envío de mensajes ya existe en el sistema, la conversión de referencias en capacidades simplemente hace que esa transparencia siga existiendo para la protección. La movilidad no se ve afectada en absoluto por la existencia de protección en el sistema.

12.2.3 Uniformidad en la Orientación a objetos

Una de las principales ventajas del modelo de protección diseñado es precisamente su modo de integración con el modelo de objetos del sistema. La protección está integrada en el modelo de objetos del sistema de forma uniforme. Tan sólo hacen falta pequeños cambios incrementales en el modelo de objetos y leves modificaciones en el comportamiento interno del sistema que no cambian la funcionalidad externa del mismo. Sólo es necesario añadir una nueva instrucción para la protección. El mecanismo de protección es transparente al resto del sistema. Una prueba de ello es que las aplicaciones existentes funcionarán sin ninguna modificación exactamente igual que en el mismo sistema sin protección. El sistema presenta el mismo comportamiento que tenía, salvo cuando se utilizan la nueva instrucción de restricción de permisos para la gestión de la protección.

12.2.4 Homogeneidad

El mecanismo propuesto cumple con la exigencia de homogeneidad que se solicitaba. Sólo es necesario este mecanismo de protección en el sistema, y todos los recursos que existen en el sistema se protegen de la misma manera con el mecanismo. No se establecen diferentes mecanismos de protección dependiendo del recurso que se pretenda proteger, como en sistemas tradicionales (memoria, ficheros, etc.).

En el sistema la única abstracción que existe (recurso) son objetos, y este mecanismo de protección protege a todos los objetos por igual, sin distinciones, independientemente del tamaño de estos objetos. La uniformidad y homogeneidad del sistema en torno a la orientación a objetos se extiende también a la protección.

12.2.5 Protección de granularidad fina

Otro de los requisitos impuestos al diseño fue el hecho de que la protección fuera de granularidad fina. Es decir, deben protegerse individualmente cada una de las operaciones de un objeto (se consigue con los permisos de longitud variable de las capacidades), y debe poder especificarse permisos individuales sobre estas operaciones para cada uno de los posibles objetos cliente (cada objeto tiene sus capacidades particulares para acceder a otros objetos). En realidad puede verse este requisito como una manera de enunciar el principio de mínimo privilegio. En el sistema, como ya se mencionó anteriormente, se puede seguir el

principio de mínimo privilegio debido a que se dispone de un nivel de granularidad de protección lo más fino posible.

12.3 Resolución de problemas de un sistema de seguridad

12.3.1 Defensa perimetral

La defensa perimetral del sistema queda fácilmente asegurada. En este apartado de la defensa perimetral el objeto servidor (recursos del sistema, en general) no tiene ningún canal de comunicación con el espía (atacante), o lo que es lo mismo, no existe ninguna capacidad que desde el servidor referencia al atacante. Es totalmente seguro conceder acceso al objeto privado al servidor, simplemente pasándole una capacidad de acceso al objeto al servidor, como parámetro del método correspondiente del objeto servidor.

El atacante está completamente imposibilitado para acceder al objeto privado puesto que le es imposible en esta situación obtener una capacidad de acceso al mismo.

12.3.2 Delegación (representante confuso)

Como ya se mencionó en el capítulo 9, el problema de la delegación también queda resuelto fácilmente. Cuando un objeto servidor necesite utilizar algún objeto privado en representación de otros objetos (como en el caso del compilador y el fichero de facturación), simplemente se le pasa una capacidad de acceso al mismo como parámetro de un método del servidor dispuesto al efecto. Previamente se restringe la referencia para conceder al servidor exactamente la mínima autoridad que necesite para desempeñar su trabajo. El objeto servidor (compilador) accederá al objeto únicamente a través de la capacidad y de la manera adecuada.

Los otros objetos (espía) no podrán confundir al servidor (compilador) para estropear el objeto privado, puesto que para ello necesitarían tener de antemano una capacidad de acceso al mismo para poderla pasar como parámetro al compilador (tampoco lo necesitarían puesto que si ya tienen acceso al objeto privado no necesitan confundir a nadie. Como es evidente, el único objeto que tiene inicialmente la capacidad de acceso al objeto privado es el cliente, y nunca la cedería a estos objetos espía.

12.3.3 Confinamiento

El problema del confinamiento requiere una solución más avanzada para el caso complejo, como se adelantó en el capítulo 9. En el caso sencillo se trata de asegurar que no existe un canal de comunicación (una referencia) de un objeto servidor a un objeto espía con el que está compinchado. Esto se logra fácilmente examinando la estructura de los objetos (lo que la reflectividad de la máquina abstracta permite fácilmente) para comprobar que el servidor no tiene referencias hacia objetos espía a los que no debe acceder.

En un caso más complejo, puede garantizarse el confinamiento incluso aunque el servidor tenga un referencia hacia un objeto espía. En este caso hay que controlar que no pueda utilizar esa referencia para invocar un método del objeto espía pasando como parámetro la capacidad de acceso al objeto privado. Es decir, se necesita un monitor de referencia que permita mediar en las invocaciones a métodos para comprobar que se cumplen las normas de confinamiento que se establezcan en cada momento.

De nuevo la reflectividad de la máquina abstracta y la orientación a objetos permiten implementar esta funcionalidad del monitor de referencia con flexibilidad. La máquina debe exponer reflectivamente la parte de la misma que se encarga del mecanismo de envío de mensajes. Así, cuando se envíe un mensaje a un objeto, este conjunto de objetos reflectivos

puede (si se desea implementar la funcionalidad del monitor de referencia) llamar a un objeto de usuario que implemente un monitor de referencia. Este se encargaría de comprobar los parámetros, a qué objetos apuntan estas capacidades, etc. (aprovechando de nuevo la reflectividad de la máquina abstracta). En caso de que se cumpliera el confinamiento, cedería el control al resto del mecanismo de invocación de métodos. En caso contrario, propagaría una excepción al objeto que inicia la invocación.

La reflectividad permite una implementación flexible por código de usuario de un monitor de referencia, que sólo se incluye en el sistema cuando su funcionalidad sea necesaria. Además, puede cambiarse su implementación por otra más eficiente cuando se desee, de manera dinámica, como código normal de usuario que es. Puede crearse una jerarquía de clases para implementar monitores de referencia, especializando su comportamiento en función de las necesidades. Así, podrían utilizarse monitores que además del confinamiento, permitieran la trazabilidad en el uso de las capacidades, la revocación selectiva proporcionando un nivel de indirección a través de fachadas, etc.

12.4 Ventajas adicionales

12.4.1 Protección automática de capacidades

El diseño de capacidades que aquí se propone, donde éstas se localizan en el nivel más interno, dentro de la máquina, combina las ventajas de las capacidades hardware y también de las capacidades dispersas.

Por un lado, al igual que las capacidades segregadas o etiquetadas, se asegura la imposibilidad de falsificación de una capacidad, puesto que la máquina ofrece un lenguaje tipado en el que sólo es posible la invocación de métodos, pero nunca el acceso directo a los atributos de los objetos. En el caso de las capacidades, los permisos son atributos de la referencia, y el único modo de acceder a ellos es a través de las operaciones que ofrecen éstas, y nunca puede cambiarse su valor arbitrariamente.

Por otro lado, al igual que las capacidades dispersas, las capacidades se utilizan directamente en las estructuras de usuario (son las referencias normales que usa la máquina en estas estructura). Esto proporciona una gran flexibilidad, ya que es el usuario el que puede, mediante operaciones comunes propias de su trabajo de usuario, (y no con operaciones especializadas del sistema operativo, por ejemplo) gestionar las capacidades: crearlas, restringirlas y transferirlas.

12.4.2 Permisos de longitud arbitraria

Muchos sistemas presentan capacidades con una cadena fija de bits para identificar permisos de las operaciones. De este modo sólo pueden protegerse un número fijo o predefinido de operaciones. La semántica completa de los objetos requiere que todos y cada uno de los métodos de un objeto sean protegidos.

La incorporación de las capacidades como nuevas referencias de la máquina permite que la longitud de los permisos sea igual a la que necesita cada objeto (número de métodos que tenga cada objeto). Las referencias que se tratan como una abstracción más del sistema, cuya implementación física no se ve en los objetos de usuario. Es decir, un objeto de usuario contiene referencias, pero no existe un tamaño físico reservado para las mismas que se vea en el espacio de usuario. Esto permite que la implementación interna de las mismas reserve el número de permisos necesario para cada objeto sin afectar a los objetos de usuario.

12.4.3 Extensibilidad del sistema con seguridad y flexibilidad

El conjunto del sistema junto con el mecanismo de protección permite que la funcionalidad del sistema se extienda de manera flexible (como en el caso de la incorporación de un monitor de referencia, descrito anteriormente, o cualquier otra funcionalidad).

El mecanismo uniforme de protección hace que esta extensión pueda hacerse de manera controlada y segura. Puesto que las extensiones se hacen mediante objetos normales (como es el sistema en su conjunto), se usan y protegen de la misma manera que cualquier otro objeto: mediante capacidades, que se pueden pasar a los objetos adecuados y restringidas previamente de manera adecuada. No es necesario introducir mecanismos específicos para controlar la extensión del sistema, el mecanismo único es suficientemente general.

El sistema se puede proteger, por un lado, de comportamiento inadecuado de estas extensiones, aplicando el principio de mínimo privilegio que permiten las capacidades, simplemente pasando a los objetos que extienden el sistema únicamente las capacidades para los objetos que necesiten acceder, y con los permisos adecuados.

De la misma manera se controla qué elementos se pueden extender en el sistema y por quién, estableciendo qué objetos tienen capacidades de acceso a los objetos que permiten incorporar las extensiones, y con qué permisos. Por ejemplo, la instalación del monitor de referencia se podría llevar a cabo con una llamada a un objeto del sistema (que representara reflectivamente a la máquina abstracta) de este estilo: `system.instalarMonitor(nuevoMonitor)`. Bastaría con desactivar esta operación en las capacidades que se pasaran a los objetos para que pudieran acceder a este objeto del sistema. Únicamente se pasaría una capacidad para este objeto del sistema con la operación activada a aquellos objetos a los que se permite instalar monitores de referencia.

12.4.4 Abstracciones adicionales para la protección innecesarias

No se necesitan otras abstracciones para introducir la protección en el sistema que las propias capacidades (que a su vez son una ampliación del concepto existente de referencias). No es necesaria la existencia de conceptos adicionales como usuario, dominio, etc. para que el sistema funcione de manera segura.

Sólo en aquellas aplicaciones que necesiten utilizar estas abstracciones, éstas se pueden introducir fácilmente como clases y objetos normales de usuario (véase el capítulo 18). Pero no forman parte del núcleo básico del sistema, y, por tanto, las aplicaciones que no las necesiten no se ven obligadas a pagar su costo sin necesidad.

12.4.4.1 Concepto de superusuario innecesario: sistemas más seguros

Así por ejemplo, no se necesita el concepto de un usuario omnipotente que tiene autoridad sobre todos los elementos del sistema (superusuario[1]). Como es bien conocido, esto es causa de enormes problemas de seguridad en los sistemas que tienen estos conceptos, normalmente ligados además a la protección con listas de control de accesos. Lamentablemente, en estos sistemas, a través de múltiples agujeros de seguridad puede verse comprometido el acceso al superusuario. Por ejemplo por el fallo en la implementación de un servicio que funcionaba bajo ese usuario. Una vez que se accede por alguno de estos caminos al superusuario, o a poder ejecutar programas en su nombre, la seguridad de todo el sistema está comprometida (puesto que el superusuario tiene poderes absolutos), y no únicamente la de ese usuario en particular (que sería lo que el sentido común indicaría como razonable).

[1] Aunque siempre podría implementarse esta funcionalidad si se necesitase.

13 Políticas de Seguridad sobre el Mecanismo Básico

Como se ha dicho ya en capítulos anteriores, uno de los requisitos, y por tanto, una de las ventajas, que se imponen al mecanismo de seguridad ideado, es la flexibilidad del mismo. Gracias a su diseño simple, es posible implantar diferentes políticas de seguridad por encima de él, que estén ajustadas a las necesidades concretas del sistema de computación en el que se trabaje, sin necesidad de lastrar el sistema con funcionalidad no utilizada en todos los escenarios posibles.

En un SIOO existirá un conjunto de objetos sobre la máquina base, que conforman el Sistema Operativo. Entre algunas de las funciones que se le atribuyen al Sistema Operativo (en el caso de un sistema tradicional multipropósito) se encuentran, por ejemplo, el manejo de usuarios, su conexión al sistema, y la creación de un entorno de computación (*shell*) personalizado.

Asimismo, normalmente el Sistema Operativo se encargará de mantener un "Servicio de Denominación", o "Servidor de Nombres", que gestiona los nombres simbólicos de todos aquellos objetos que lo tengan. Se trata de un concepto similar al "Servidor de Ficheros" de los sistemas clásicos, pero más amplio, puesto que en este último los ficheros son las únicas entidades que pueden ser denominadas en el sistema, y en un sistema integral serán objetos de cualquier clase.

Cuando un usuario accede al sistema, contará con la posibilidad de acceso al servicio de denominación, que le proporcionará los objetos sobre los que tenga derecho de acceso. A través de este servicio, el usuario puede pedir acceso a través de su nombre simbólico y el servicio le devuelve su referencia/capacidad para que pueda utilizar el objeto. De esta manera podrá trabajar con objetos de forma análoga a como se trabaja en los sistemas clásicos, donde el usuario puede lanzar aplicaciones y utilizar ficheros a través de su nombre simbólico.

Con el uso de un mecanismo de protección como el que aquí se propone, existen diferentes formas de gestionar la seguridad en un entorno de computación de este estilo. En los siguientes apartados se presentan algunas de las diferentes posibilidades, lo que corrobora la flexibilidad del mecanismo.

Aunque no es éste objetivo fundamental de esta tesis doctoral, en este apartado se van a esbozar algunos aspectos relativos a políticas de seguridad que podrían implantarse sobre el mecanismo básico, y que resuelven diferentes situaciones a las que puede estar sometido el sistema.

En primer lugar se plantea una situación en la que el sistema debe soportar el concepto de usuario, el cual se comunica con el sistema a través del tradicional intérprete de órdenes. Asimismo el sistema cuenta con un servicio de denominación que gestiona los objetos con nombre, y que son propiedad de los diferentes usuarios. En esta situación se presentan diversos enfoques, que serán analizados.

Por otro lado, se presenta una situación en la que el sistema requiera el manejo de revocación de autoridad cuando el usuario así lo desee, que afectará a los objetos con nombre simbólico manejados por el usuario.

En tercer lugar, se describe una situación en la que el sistema debe controlar la propagación de autoridad, mediante el uso de un monitor de referencia que utiliza una política de este tipo.

13.1 Estructura general del sistema.

Se trataría de un entorno en el que se pudieran conectar los usuarios, y en el que cada usuario tuviera un espacio privado donde residirían los objetos creados por el usuario. Podrían existir otros espacios a los que tuvieran acceso compartido más usuarios.

Se puede plantear un sistema en el que un objeto *login* lee el nombre y la clave de usuario, y comprueba con una tabla de usuarios (que contiene objetos que representan a cada usuario) que el usuario existe y la clave coincide. Esta información, junto con un servidor de nombres privado (espacio de trabajo privado) en el que cada usuario registra los objetos que crea, será parte de los atributos del objeto usuario.

Posteriormente se crea un objeto *shell*, inicializándolo (pasando la referencia) con el servidor de nombres privado del usuario, y a partir de aquí el usuario puede realizar operaciones, invocando objetos existentes, creando objetos nuevos a partir de clases existentes, visualizando los objetos a los que tiene acceso en el servidor de nombres, restringiendo capacidades de objetos, o enviándolas a otros usuarios.

En un SIOO resulta necesario poder contar con objetos que tienen asignado un nombre simbólico y a los que se puede acceder cuando sea necesario. El usuario, por ejemplo, desde el *shell* que le ofrece el sistema, puede querer lanzar un objeto o crear un objeto de una clase existente de la que sólo sabe su nombre simbólico. Por otra parte, dentro de la ejecución de un objeto, en ocasiones resulta útil acceder a un objeto o clase de la que sólo se conoce su nombre simbólico.

Para ello, como parte de los objetos que integran el sistema operativo, aparece el objeto "servidor de nombres" que proporciona el servicio de denominación, cuya función consiste en devolver una referencia/capacidad a un objeto cuando recibe el nombre simbólico del objeto como parámetro de una operación del servidor que conecta el nombre con su referencia. Adicionalmente debe permitir registrar y eliminar ("dar de alta y de baja") objetos en el servicio.

13.1.1 Objeto servidor de nombres

Asocia nombres simbólicos con referencias a objetos (nombre, referencia). Es análogo al servidor de ficheros tradicional, aunque más genérico, pudiendo almacenar cualquier tipo de objeto.

Los métodos pueden ser:

- Buscar(nombre): referenciaRetorno. Dado un nombre simbólico, esta operación devuelve la referencia del objeto asociado al nombre.
- Registrar(nombre, referenciaFuente). Permite añadir un nuevo objeto (indicando su referencia) al servidor de nombres, asociándole un nombre simbólico.
- Eliminar(nombre). Permite eliminar un objeto del servidor de nombres. En el caso de que el servidor fuera compartido por diferentes usuarios (por ejemplo un servidor público, al estilo de un tablón de anuncios) se necesitaría algún control adicional para que sólo pudiera borrarlo el usuario que lo insertó. Por ejemplo, puede hacerse que el servidor almacenara además quién fue el usuario que insertó la referencia (añadiendo

la referencia del objeto usuario correspondiente, que sería un dato más que habría que pasar al iniciar el *shell*). Inicialmente se puede dejar que cualquiera pueda borrar, puesto que para servidores privados únicamente tiene una referencia de acceso el usuario propietario.

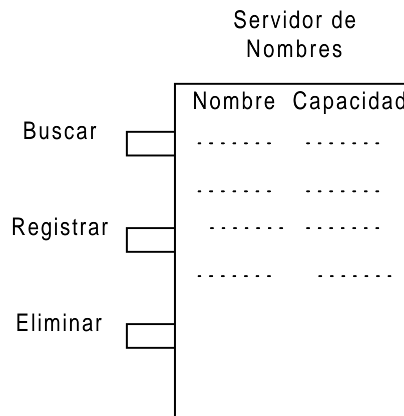


Figura 13.1: Objeto Servidor de Nombres

13.2 Políticas de Seguridad

Con la estructura anterior se pueden plantear diferentes modos de gestionar el espacio de objetos visible por un usuario. Como se verá a continuación, la política de seguridad puede ir ligada a la gestión del servicio de denominación que ofrece el sistema operativo. El conjunto de objetos que el servidor de nombres deja visible a un usuario constituyen el dominio de protección de un usuario. Es posible plantear diferentes dominios, que van desde los más restringidos a los más flexibles. Todos ellos emplearán el mecanismo básico de protección en el nivel más bajo.

13.2.1 Dominios privados

El caso más sencillo consiste en asignar un servidor de nombres diferente para cada usuario. La clase servidor contará con tantas instancias como usuarios haya en el sistema y en cada una se almacenan los nombres de objetos accesibles para un usuario. El servidor de nombres de cada usuario puede verse como un dominio privado, puesto que en él está el conjunto de objetos que pueden ser accedidos por el usuario

La única "política" existente, consiste en que a cada *shell* se le pasa un servidor de nombres (dominio, conjunto de objetos a los que puede acceder) que depende del usuario que se trate y que será el único al que el usuario pueda acceder, puesto que será la única referencia a un servidor de nombres con que se inicializará el *shell* que se cree para el usuario.

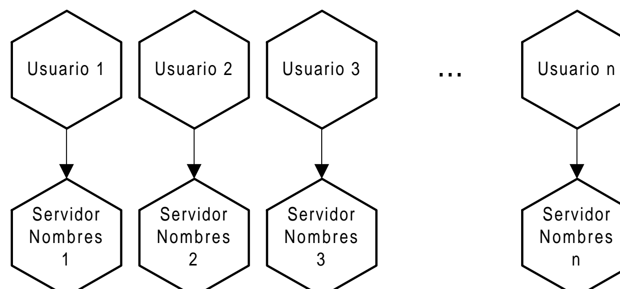


Figura 13.2: Dominios privados

13.2.2 Dominios públicos

Otra posibilidad podría ser la de contar con un dominio privado para cada usuario y un dominio público compartido por todos. Cuando un usuario entra en el sistema, se le asocian dos dominios, el privado donde guarda sus objetos personales, y el público, donde tiene acceso a objetos a disposición de todos los usuarios. El dominio público es un servidor de nombres igual que los otros, y devuelve la referencia/capacidad para los objetos públicos. Esta referencia/capacidad tendrá ciertos permisos sobre los objetos, los establecidos previamente por el usuario que la insertó en el servidor. Cualquier usuario que pida al servidor un mismo nombre, recibirá la misma capacidad, y por tanto tendrá los mismos permisos sobre el objeto.

13.2.2.1 Un único servidor público

El servidor de nombres público tendrá operaciones para insertar o borrar objetos de él, y todos o algunos usuarios podrán recibir capacidad para esta operación, con lo que dinámicamente pueden dar de alta nuevos objetos de forma pública. El servidor funciona como un tablón de anuncios en el que cualquier usuario puede utilizar los objetos que se expongan en el tablón (con los permisos establecidos por el usuario que insertó el objeto en el tablón). Adicionalmente, se puede hacer que sólo ciertos usuarios tengan posibilidad de añadir objetos al tablón, simplemente eliminando los permisos para estas operaciones antes de pasar la referencia al servidor público a los usuarios que no puedan insertar contenido.

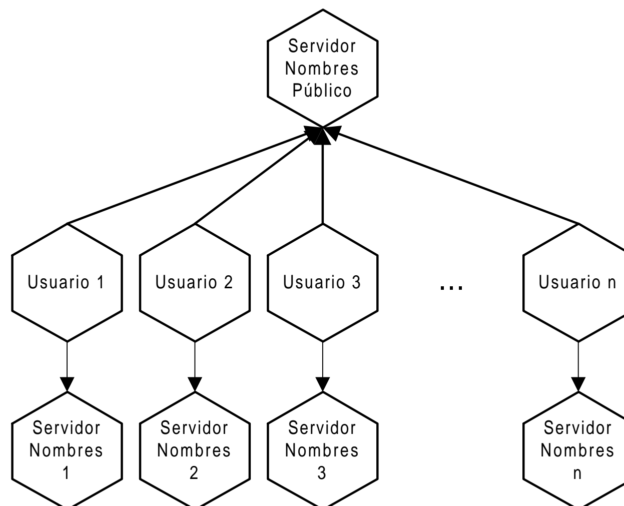


Figura 13.3: Dominio público

13.2.2.2 Buzones para ampliar las capacidades de los usuarios

Si bien este modelo es más completo que el anterior en el que únicamente existen dominios privados, todos los usuarios tienen exactamente los mismos permisos sobre los objetos declarados como públicos. En ocasiones, esto no es suficiente, dado que normalmente se desea que existan permisos diferentes sobre el mismo objeto dependiendo del usuario. Un modo de proceder podría ser permitiendo que se insertasen objetos (capacidades en realidad) en el dominio privado del usuario deseado, previamente habiendo restringido de manera adecuada al usuario los permisos de acceso al objeto.

Una variante es que exista un dominio adicional por cada usuario, que actúe a modo de buzón público. En este buzón cualquier usuario podría depositar capacidades restringidas para que el propietario del buzón accediera a esos objetos. De esta manera el dominio privado seguiría siendo utilizado en exclusiva por su propietario. Esta estructura da soporte,

directamente, a un sistema de correo electrónico entre los usuarios. Es más amplio, sin embargo, puesto que en lugar de enviarse a los buzones mensajes simplemente de texto, se puede enviar cualquier objeto.

En cualquier caso, existen otras posibilidades adicionales.

13.2.3 Dominios asociados a clases

En las propuestas anteriores, los dominios van asociados a los usuarios. Sin embargo, una política más abierta sería aquella que asocia los dominios a las "aplicaciones". Resulta deseable, para cumplir el principio de mínimo privilegio, que cada aplicación tenga sólo acceso a los métodos de los objetos estrictamente necesarios.

Una posibilidad sería la creación de objetos "política de seguridad" que asocien un nombre de clase (aplicación) con un dominio de objetos a los que pueda acceder esa clase (aplicación) si se usa esta política de seguridad. Por ejemplo:

Objeto política de seguridad P:
- Clase Editor - Dominio A (ficheros en modo lectura / escritura)
- Clase Impresora - Dominio B (Impresoras en modo escritura)

La clase política contendría una tabla con los campos "nombreClase", refDominio (referencia a un servidor de nombres). De esta manera, cualquier editor que cree el usuario sólo podrá acceder al conjunto de ficheros (objetos documento) que se encuentren dentro del dominio A (en el cual se habrán colocado en modo lectura/escritura, aunque cada documento podría tener los permisos individuales adecuados).

Cada usuario podría recibir una política propia, de forma que usuarios diferentes puedan realizar operaciones diferentes sobre diferentes conjuntos de objetos mediante la misma la misma aplicación.

En realidad la política puede verse como un caso particular de un servidor de nombres, que asocia los nombres de las clases con un dominio apropiado según las normas que determine la política (puede usarse herencia, si es que luego se necesita ampliar la semántica de las políticas) para añadir más operaciones que las estrictamente correspondientes a un servidor de nombres.

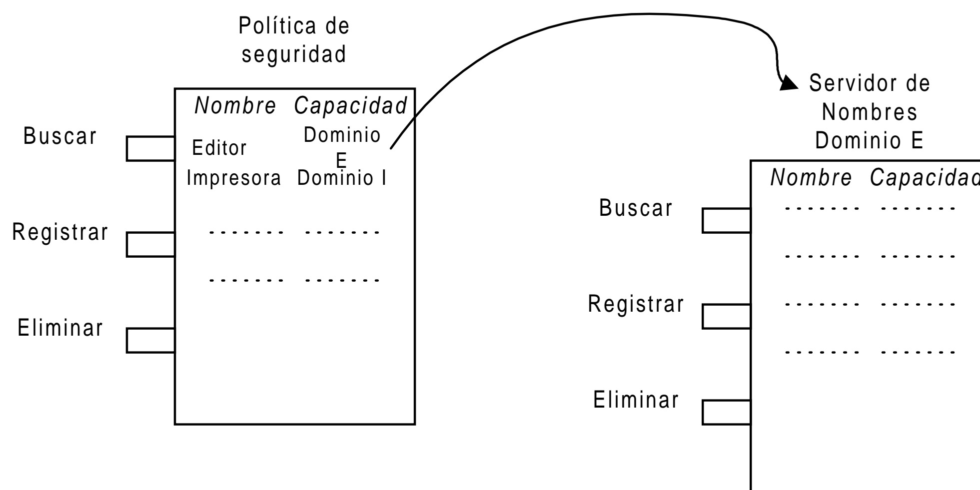


Figura 13.4: Dominios asociados a clases (aplicaciones)

Este caso tiene una funcionalidad y un comportamiento similar a la arquitectura de seguridad de la plataforma Java (JDK1.2) que se mencionó en el capítulo 6. Es una muestra de la flexibilidad del mecanismo de protección propuesto, puesto que se puede conseguir la funcionalidad de otros sistemas sin necesidad de introducir en la funcionalidad fija en la plataforma base. Simplemente se utilizan la creación normal de clases de usuario para introducir funcionalidad adicional, pero sólo en el caso de que se necesite.

13.2.4 Dominios arbitrarios

De nuevo es posible ampliar el sistema, haciendo que cada usuario pueda contar con diferentes políticas, en lugar de con una única política. Estos diferentes objetos política, pueden colocarse a su vez con un nombre en un servidor de nombres que agrupe todas las políticas (bien privado, bien un servidor público del sistema), y el usuario elegirá una u otra en función de las necesidades. En realidad no es necesario utilizar un servidor de nombres separado, puesto que los objetos política, como cualquier otro objeto, pueden insertarse en un servidor, mezclados con otros objetos. Por ejemplo:

```
Servidor de nombres para políticas:
 - "política restrictiva", refObjetoPolíticaRestrictiva
 - "política permisiva", refObjetoPolíticaPermisiva
```

La política restrictiva podría ser la anterior y la permisiva podría tener las mismas clases, o más que el objeto anterior y en los dominios de cada clase, el conjunto de objetos que aparecen y sus permisos sería mayor (más permisivo).

El usuario puede así escoger qué política aplicar a las clases que instancia, obteniendo la política adecuada de un servidor de nombres. Evidentemente, como en el caso del *shell*, las aplicaciones deben estar codificadas para que se inicialicen con un dominio de objetos determinado a los que se puede acceder (espacio de trabajo).

Como ejemplo de un posible uso se da el que sigue. El usuario tiene almacenados en un servidor de nombres privado (snp) las políticas anteriores, y quiere lanzar un programa (clase Editor) que le descarga dinámicamente de la red. Podría hacer algo así:

```
e Editor, p Política, d dominio, snp ServidorNombres
// Coger una política adecuada del servidor de nombres (será el objeto p)
snp.Buscar("política restrictiva porque viene de internet, cuidado"): p;
// Obtener el dominio d que la política (restrictiva) asocia a la clase
// Editor (podría equivaler a Buscar en el servidor de nombres)
p.getDominio("Editor"): d;
new e // Crear un editor e
e.Run(d); // Ejecutarlo con el dominio restringido adecuado
```

Es evidente que previamente hay que rellenar de manera adecuada los dominios y las políticas correspondientes, lo cual puede realizar de manera automática un elemento del sistema que se ocupe de controlar la seguridad, basándose, por ejemplo, en una especificación genérica de las políticas de seguridad que se desean.

El procedimiento anterior, lógicamente, también se automatizaría para la mayoría de los usuarios, encapsulándolo en una única operación de un objeto de utilidad, o como parte de un guión (*script*) o función de utilidad del intérprete de órdenes.

El conjunto de políticas que se pueden utilizar puede guardarse en un servidor de nombres (directorio) aparte que contenga sólo políticas de seguridad.

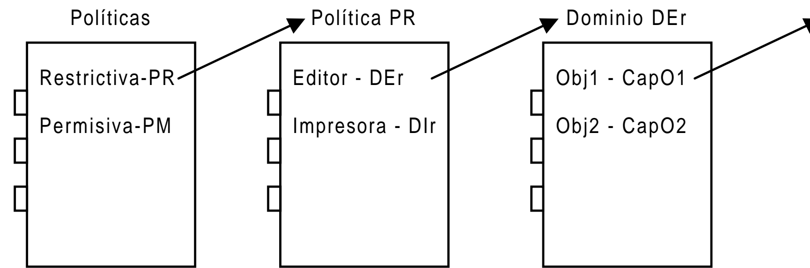


Figura 13.5: Dominios arbitrarios

13.2.5 Políticas asociadas a usuarios

Una variante de lo anterior utilizaría un gestor de seguridad en el que se almacenen únicamente las políticas de seguridad que puede utilizar un determinado usuario, con dominios rellenados por una instancia superior del sistema colocando los objetos globales que pueden ser accedidos por cada usuario. Como parte de los datos de un usuario podrían estar las políticas de seguridad (el directorio de políticas) que el sistema le deja usar.

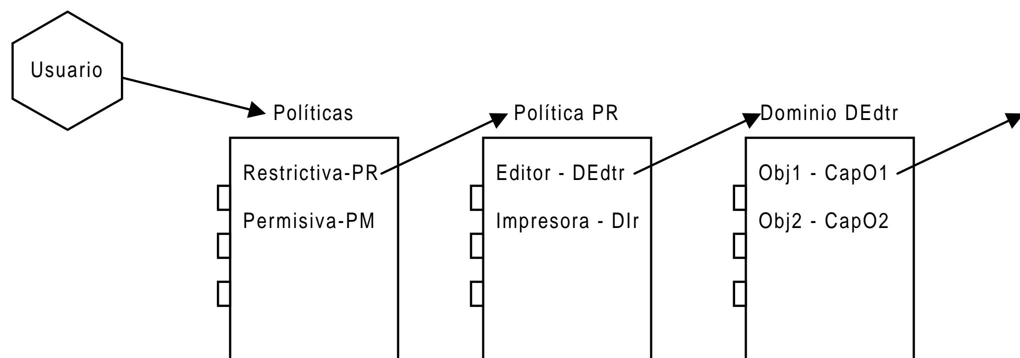


Figura 13.6: Políticas asociadas a usuarios

13.2.6 Dominios gestionados por un gestor de seguridad

Otro aspecto que habría que controlar ahora que pueden aparecer objetos globales o comunes del sistema en las políticas (en los dominios), es que el usuario no pida un dominio a la política de seguridad y luego lo utilice con una aplicación distinta de la aplicación para la que fue pedido el dominio. Por ejemplo antes se podría hacer esto:

```
p.getDominio("Telnet"): d;
new e // Crear un editor
e.Run(d);
// Se ejecuta con el dominio que asociado al TELNET, no al editor
```

Con lo que un editor podría usar las conexiones de red que estuvieran permitidas para el telnet, por ejemplo. Es decir, el usuario puede (conscientemente o por error) saltarse la política de seguridad asignada a cada clase.

Podría modificarse el modo de funcionamiento de la política para que el propio "gestor de seguridad" o política instanciase la clase y le asociase el dominio en lugar de hacerlo el usuario:

```
p.creaAplicacion("Telnet"): t;
// instancia aplicación, asocia dominio correspondiente según política
...
t.funcionar();
```

El usuario nunca tendría acceso a los dominios contenidos en las políticas (salvo que los hubiera creado él en un servidor propio), puesto que no se le daría acceso directo al usuario al servidor de políticas. En ningún caso el usuario podría ganar acceso a objetos prohibidos para una aplicación, puesto que en caso de crear un dominio propio sólo podrá colocar en él objetos a los que tuviera acceso directamente por derecho propio como usuario.

De esta manera se consigue que existan en el sistema un conjunto de aplicaciones que puedan utilizar recursos especiales del sistema con seguridad, y se pueden crear políticas particulares con más o menos permisos dependiendo de la importancia del usuario.

Esto necesita que la política de seguridad "sepa" cual es el método inicial de una aplicación para asociarle el dominio, aunque podría imponerse un determinado convenio, al estilo de la cabecera public static main del método inicial de una clase en Java.

13.2.7 Ejemplo de política con revocación

Otro de los aspectos ya mencionados, que en ocasiones resultan interesantes, dentro de un sistema de computación es el hecho de la posibilidad de revocación de permisos después de haber sido realizada la concesión. En este apartado se mostrará un ejemplo de cómo la política puede solucionar este problema.

Supóngase una situación, análoga a las anteriormente planteadas, con usuarios que acceden al sistema a través de un *login* y un *shell*, y que cuentan con servicios de denominación para almacenar nombres simbólicos de objetos.

Supóngase que un usuario, que en un instante dado concede una capacidad a otro sobre un objeto, posteriormente decide retirar todos o parte de los permisos. Esta situación no se resuelve simplemente eliminando el objeto del servidor de nombres correspondiente, puesto que el usuario que lo recibe puede haber recibido ya la capacidad, y por tanto no es posible saber qué uso hace de ésta.

Para solucionar totalmente el problema, es posible utilizar un mecanismo de indirección, basado en la creación de fachadas que se interponen entre el usuario y el objeto real. Este mecanismo ya ha sido comentado en capítulos anteriores. Basta con crear un gestor de fachadas que se encargue de crearlas y eliminarlas cuando un cliente se lo indica.

Si un usuario A desea conceder una capacidad con revocación a otro B, en lugar de insertar ésta directamente en el servidor de nombres de B, previamente invoca al gestor de fachadas para que cree una a partir de la capacidad al objeto que le indica el usuario A (y que poseía previamente).

El gestor de fachadas crea un objeto, que tiene la misma interfaz que el objeto real, pero cada operación consiste en una redirección, invocando a la misma operación del objeto real. El gestor de fachadas devuelve la capacidad para la fachada al usuario A, el cual la introduce en el servidor de nombres del usuario B.

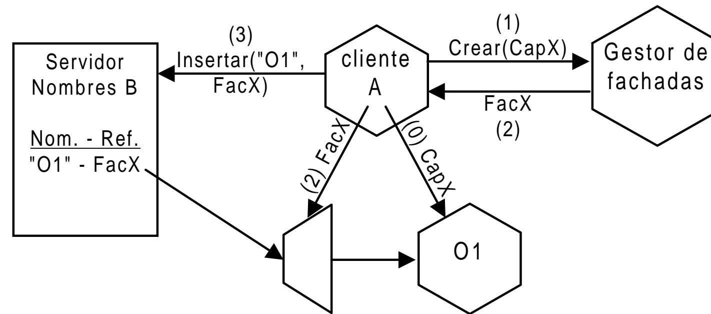


Figura 13.7: Acceso indirecto para revocación mediante fachadas y un gestor.

Si posteriormente A quiere eliminar la capacidad concedida a B, bastará con que éste elimine la fachada, bien eliminándola como objeto normal, bien indicándole a la propia fachada que se elimine (u otras operaciones si las fachadas fueran más inteligentes, por ejemplo podrían contar el número de veces que se utilizan, etc.). Este procedimiento puede ser también automatizado por un objeto gestor

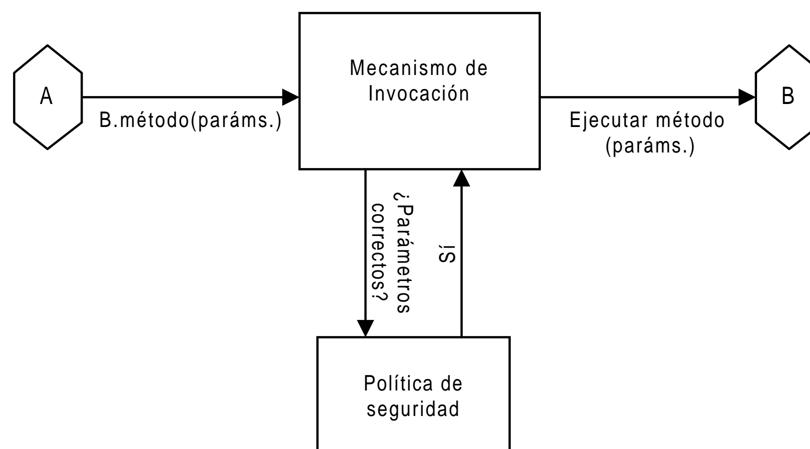
13.2.8 Ejemplo de política con control de propagación

Otro aspecto interesante en un sistema de seguridad puede ser el control de la propagación de autoridad. Este control lograría el confinamiento.

Supóngase que se crea un nuevo servidor, por ejemplo la impresora, que cuenta con una operación de impresión disponible para los usuarios "normales". Ni cliente ni servidor confían el uno en el otro, por lo que ambos deben establecer restricciones a la hora de pasar parámetros de entrada y salida.

Supóngase que el método imprimir necesita dos parámetros: uno de entrada, con la capacidad para leer los datos del objeto a imprimir, y otro de salida, donde se almacenan posibles errores, que será también de lectura por parte del cliente.

Cuando el cliente llama a imprimir, en la invocación al método deberá comprobarse que los permisos de los parámetros son los correctos. Para ello, habrá que modificar ligeramente el mecanismo de invocación de métodos de manera reflectiva, que como se sabe, en el Sistema Integral Orientado a Objetos planteado, se localiza dentro de la máquina. En la modificación se trata de realizar la comprobación de permisos, mediante la consulta de la política de seguridad, que indica cuáles son los permisos que deben contener los parámetros pasados.



14 Especificación de la Máquina Abstracta sobre la que se Implanta el Mecanismo Básico de Protección

En el capítulo 11 se realiza una descripción detallada de las características deseables de un mecanismo de protección, razonando el por qué de la elección realizada. Todas estas ideas allí desarrolladas, se han llevado a la práctica mediante un prototipo dentro de un Sistema Integral Orientado a Objetos, para, de esta manera, comprobar la bondad del diseño y demostrar que esta nueva concepción de la protección de un sistema aporta y combina ventajas sobre otras soluciones existentes.

Partiendo de una versión existente en la que no existía ningún mecanismo de protección, se modificó ésta para añadir el mecanismo y comprobar las implicaciones de rendimiento que envuelven su inclusión. Asimismo, se implementó un entorno operativo para la demostración de la flexibilidad del mecanismo y las ventajas aportadas por éste.

Para poder realizar una descripción detallada del mismo, es necesario comenzar por describir el sistema Oviedo3 en su conjunto, y la máquina abstracta que forma parte de él, puesto que ésta será la localización del mecanismo básico, como se decidió en el diseño realizado.

14.1 El sistema Integral Orientado a Objetos Oviedo3

Oviedo3 [CIA+96] es un proyecto de investigación desarrollado por el grupo de investigación en Tecnologías Orientadas a Objetos de la Universidad de Oviedo, que constituye un diseño de un Sistema Integral Orientado a Objetos, y por tanto sigue la arquitectura mostrada en el capítulo 5 de resumen de la arquitectura de un SIOO.

14.1.1 Esquema general del sistema Oviedo3

Los elementos básicos que proporcionan un espacio de objetos al sistema integral son la máquina abstracta Carbayonia [CIA96], que implanta el modelo único de objetos del sistema y el Sistema Operativo SO4 [ATA+97], que extiende las capacidades de la misma. Ambos definen un sistema donde el concepto de objeto, de cualquier granularidad, constituye la única abstracción. La combinación de estos dos elementos, forman la plataforma básica, que permite dar 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 [Álv98].

El Sistema Integral Orientado a Objetos: Oviedo3

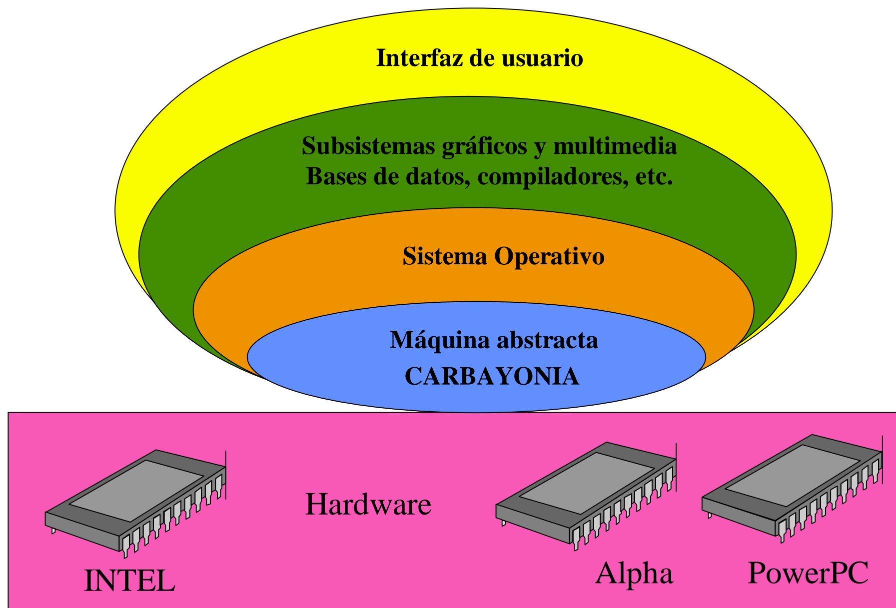


Figura 14.1: Arquitectura del sistema integral Oviedo3

14.2 La máquina abstracta Carbayonia

El modelo básico de objetos del sistema está soportado por la máquina abstracta Carbayonia, que sigue la arquitectura de referencia de una máquina de soporte de SIOO que se mencionó en el capítulo 5. Esta máquina es el “microprocesador OO”, y su lenguaje máquina, totalmente OO, se denomina Carbayón. La característica principal de esta máquina es que tan sólo tiene objetos y todas las operaciones son sobre objetos [CIA96].

Todos los objetos tienen un identificador que se usa para operar con ellos exclusivamente a través de referencias. Por tanto, las acciones sobre los objetos se realizan por medio de referencias y nunca directamente. Las referencias, como los propios objetos, tienen asociado un tipo 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.

14.2.1 Estructura de la máquina

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.

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.

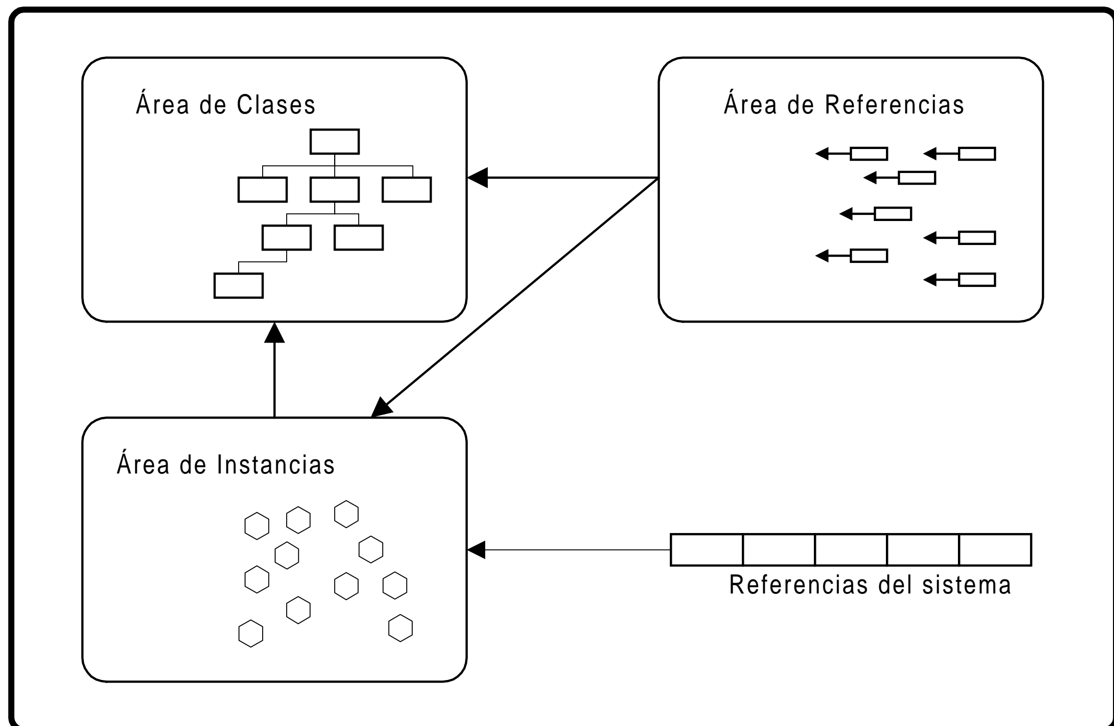


Figura 14.2: Estructura de referencia de una máquina abstracta para el sistema integral

14.2.1.1 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 procesadores 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 procesador 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.

14.2.1.2 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.

14.2.1.3 Área de Referencias

Para operar sobre un objeto es necesaria 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 asociada) 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.

14.2.1.4 Referencias del Sistema

Son una serie de referencias que están de manera permanente en Carbayonia y que tienen funciones específicas dentro del sistema.

- 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.

14.2.2 Reflectividad de la máquina como aspecto destacable

Para mantener la uniformidad en la orientación a objetos cuando se extiende la máquina con objetos del sistema operativo, se dota a ésta de una arquitectura reflectiva. La máquina es vista por el resto del sistema como un conjunto de objetos normales, que pueden libremente colaborar con otros objetos dentro del mismo paradigma. Como ejemplo, la persistencia puede ser proporcionada con objetos de la máquina colaborando con objetos del sistema operativo para guardar instancias en el disco.

14.3 El lenguaje Carbayón

El juego de instrucciones de la máquina se describirá 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. Buscando facilitar la difusión internacional, en el lenguaje se utiliza el idioma inglés. Para facilitar la comprensión de las características del lenguaje se harán comparaciones con el lenguaje orientado a objetos C++ en los lugares adecuados.

14.3.1 Instrucciones declarativas: Descripción de clases

La unidad básica declarativa de la máquina es la clase, y será la unidad mínima de trabajo que las aplicaciones comuniquen a la máquina. Las clases del modelo único tienen una serie de propiedades que hay que describir: nombre, relaciones de herencia o generalización (*isa*), relaciones de agregación (*aggregation*), relaciones de asociación genéricas (*association*) y métodos de la clase (*methods*). En Carbayonia, esto se describe como:

```
Class <Nombre>
Isa {Clase,}
Aggregation {Nombre: Clase}
Association {Nombre: Clase}
Methods
   { <Nombre>  ( {[Nombre]:[Clase]} ); }
Messages
   { <Nombre>  ( {[Nombre]:[Clase]} ); }
EndClass
```

Las clases Carbayonia tendrán una serie de características propias como son:

- No existen operadores. Se deja al compilador de mas alto nivel que simule la implementación de operadores.
- El lenguaje no tiene construcciones explícitas para denominar los métodos virtuales. En esta arquitectura, todos los métodos son virtuales.
- No se clasifican los métodos mediante *private*, *public* o *protected*. Esta labor también se deja al compilador de alto nivel.
- No hay constructores ni destructores. Estos serán también generados por el lenguaje de alto nivel.

14.3.1.1 Class (clase)

El primer elemento permite dar un nombre a la clase. Este nombre deberá ser diferente del de las otras clases y se almacenará, junto con el resto de la información que le siga, en el área de clases.

14.3.1.2 Sección Isa (herencia)

Se declaran, separadas por comas, todas las clases de las que deriva la clase que se está definiendo. Como ya se ha dicho la herencia de métodos es siempre virtual.

14.3.1.3 Sección Aggregation (agregaciones)

Aquí se enumeran los objetos que pertenecen a la clase indicando el nombre que se les da a cada uno (en realidad es el nombre de la referencia a través de la que se accederá a los mismos). La clase a la que pertenece cada uno de los objetos se indica poniendo el nombre de la clase separado por dos puntos del nombre del objeto. Se mantiene la semántica de la agregación. Los objetos agregados se crean automáticamente cuando se crea el objeto que los contiene y se destruyen cuando éste se destruye. Además, no se pueden eliminar individualmente, sólo a través de la eliminación del contenedor.

14.3.1.4 Sección Association (asociaciones)

En este lugar se declaran los objetos que participan de una relación de asociación con la clase que se está definiendo. A diferencia de los agregados, dado que son relaciones genéricas, estos objetos no se crean automáticamente al crear el objeto ni se destruyen al borrar éste. El equivalente en C++ sería declarar un puntero al tipo deseado, cuya gestión recae en el programador (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.)

Por lo tanto, con los miembros agregados puede pensarse que se guarda una instancia del objeto y con los asociados se guarda un puntero al objeto. En Carbayonia todo se hace a través de referencias por lo que para los agregados se crea una instancia a la cual apunta la referencia, y para los asociados la referencia está libre (es responsabilidad del programador hacer que apunte a un objeto creado previamente).

El conjunto de los agregados y relaciones de un objeto es el equivalente al concepto de variables o variables miembro de un objeto de otros lenguajes como C++. También se pueden llamar atributos o propiedades del objeto.

14.3.1.5 Sección Methods (declaración de métodos de la clase)

En el siguiente apartado se tratará la definición del cuerpo de los mismos. La declaración consiste en un nombre de método seguido de una serie de parámetros entre paréntesis. Cada parámetro se identifica mediante una clase a la que pertenece. Al cierre de los paréntesis se pone el tipo del valor de retorno si es que existe.

14.3.2 Características de las clases Carbayonia

14.3.2.1 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 típica en rombo, donde *B* y *C* heredan sencillamente de *A* y *D* hereda múltiplemente de *B* y *C*, 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.

14.3.2.2 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 estructura de rombo mencionada 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 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
```

14.3.2.3 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.

14.3.2.4 Uso exclusivo de enlace dinámico (sólo métodos virtuales)

No es necesario especificar si un método es virtual 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.

14.3.2.5 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.

El mecanismo de protección diseñado consigue una mayor flexibilidad al permitir 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*.

14.3.2.6 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, que ya es conocida por la máquina y se realiza automáticamente.

14.3.3 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).

14.3.4 Definición de métodos

En la declaración de una clase se declaran también los métodos que tiene. Ahora es necesario definir exactamente cuál es el comportamiento de cada método. Para ello se distingue entre la cabecera del método y el código del mismo propiamente dicho. En la cabecera se definen los parámetros del método y las referencias (variables) locales que se utilizarán en el método. En el cuerpo o código se especifican la secuencia de instrucciones que definen el comportamiento del mismo.

La estructura general de un método es la siguiente:

```
Method <Nombre>  ( {[Nombre]:[Clase]} ):[Clase]
[Refs {<Nombre>:<Clase>}]
[Instances {<Nombre>:<Clase>}]
Code
  {<Instruccion>}
EndCode
EndMethod
```

14.3.4.1 Cabecera de método

Un método se define con un nombre, unos parámetros (indicando para cada parámetro su nombre y clase) y un valor de retorno si tiene.

14.3.4.2 Refs (Referencias)

Aquí se indican todas las referencias locales que usará el método, junto con su tipo, puesto que todas las referencias tienen un tipo asociado. Al entrar en el método se crean automáticamente las referencias locales, que se pueden usar en el cuerpo para trabajar con objetos (crear objetos a partir de las referencias, invocar métodos, etc.). Al finalizar el método, estas referencias locales desaparecen automáticamente.

14.3.4.3 Instances (Instancias)

Es una cláusula similar a la anterior. La única diferencia es que la máquina gestiona automáticamente la creación y eliminación de la instancia a la que apunta la referencia. Al entrar en el método, además de la referencia, se crea la instancia a la que apunta la referencia. Al finalizar el método, se libera la instancia.

Las instancias de clases básicas pueden ser inicializadas a un valor, indicado entre paréntesis a continuación del nombre de la clase, como puede verse en el ejemplo:

```
Instances
undostres:integer(123);
 apagando:string('Apagando el sistema!');
 [...]
Code
 MiEntero.Set(undostres);
 Consola.Write(apagando);
 [...]
```

14.3.5 Code. Cuerpo del método

A continuación se describen las instrucciones que se pueden utilizar en el cuerpo de los métodos (entre las palabras Code y EndCode). Todas las instrucciones se terminan con un punto y coma. Estas son las instrucciones de comportamiento que son las que aparecen en las máquinas convencionales. Siguiendo la filosofía de disponer de un juego de instrucciones reducido, sólo existen en torno a 15 instrucciones.

14.3.5.1 Instrucciones para la creación de instancias

Se puede ligar una instancia a un objeto mediante la operación New, que crea un nuevo objeto del tipo que tenga la referencia, y deja la referencia apuntando al nuevo objeto.

```
New <Referencia>
```

En caso de que no se pueda crear el objeto, se lanza una excepción (por ejemplo por una falta de espacio).

Al crear un objeto se crean a su vez todos los objetos agregados que pertenezcan a dicho objeto. Por tanto, no se debe usar New con estos objetos agregados. Tampoco con las referencias declaradas en Instances, puesto que también se crean las instancias automáticamente. Las referencias de tipo Association se encuentran inicialmente libres (no apuntan a ningún objeto), al igual que las declaradas en Refs. Sobre estas últimas es sobre las que se puede aplicar New. Con las referencias recibidas como parámetros del método, esto depende de la lógica de aplicación que exista en cada caso.

Una restricción muy grande que tienen lenguajes como C++ es que la manera anterior de crear objetos sólo permite crear objetos de un tipo conocido en tiempo de compilación. No se pueden crear (fácilmente) objetos cuyo tipo se determine en tiempo de ejecución. Para permitir esto se puede añadir una segunda versión de la operación New, que tenga un parámetro adicional de tipo cadena (String) o derivado del que se tome el nombre de la clase de la que se creará el objeto

```
New <Referencia>, <refString>
```

14.3.5.2 Asignación de referencias

Si se tiene una referencia que apunta a un objeto se puede hacer que otra referencia apunte al mismo. Esto se hace mediante la instrucción Assign. Esta es la otra posibilidad para ligar una referencia a un objeto. Otras formas son mediante asignaciones implícitas que realiza la máquina con los parámetros de un método y el objeto de retorno.

```
Assign <ReferenciaDestino>, <ReferenciaFuente>
```

Al finalizar la instrucción, el objeto se podrá manipular por cualquiera de las dos referencias indistintamente. Se producirá una excepción si la referencia destino no es compatible con el tipo del objeto al que apunta la referencia fuente.

14.3.5.3 Invocación de métodos

Es en esencia la única cosa que se puede hacer con un objeto. El código de un método es básicamente una sucesión de llamadas a métodos.

```
<Referencia>.[<Ambito>:]<Metodo>({<Referencia>})[:Referencia]
```

La llamada al método está formada por la referencia al objeto al que se desea pasar el mensaje seguido del nombre del método. Opcionalmente éste puede ir precedido del ámbito de la referencia: uno o más nombres de la jerarquía de clases del tipo de la referencia si son necesarias para deshacer ambigüedades en los casos de herencia múltiple. A continuación van los parámetros referencia, separados por coma y finalmente, si el método tiene valor de retorno, la referencia que recogerá dicho objeto de retorno.

14.3.5.4 Eliminación de instancias

La instrucción Delete elimina el objeto al que apunta la referencia

```
Delete <Referencia>
```

Se producirá una excepción en los siguientes casos:

- La referencia está libre.
- La referencia apuntaba a un objeto que ya ha sido eliminado.
- Se intenta liberar un objeto agregado. Estos objetos no se pueden eliminar directamente. Sólo se podrán liberar cuando se libere el objeto del que forman parte, y esto lo hace automáticamente la máquina.

14.3.5.5 Control de Flujo

A continuación se relacionan las instrucciones que modifican el flujo de ejecución de la máquina.

Finalización de un método

La instrucción Exit finaliza la ejecución de un método.

```
Exit
```

Esto libera todas las referencias que se hayan creado mediante Refs y todas las referencias y objetos que se hayan creado mediante Instances.

Salto incondicional

Cambia la ejecución secuencial de instrucciones normal de un método, saltando incondicionalmente a una etiqueta específica que marca una instrucción determinada.

```
Jmp Etiqueta

...

Etiqueta: // Otras instrucciones

...
```

Salto condicional

Dirigen la ejecución del programa a otro punto dentro del mismo método marcado con una etiqueta si se cumple una cierta condición. Estas instrucciones tienen dos parámetros: el objeto sobre el que se comprueba la condición y la etiqueta a la que se salta.

- **Comprobaciones sobre objetos booleanos**. Utilizan un objeto de tipo básico Bool (booleano). Hay dos tipos de salto en función de si el objeto Bool vale verdadero o falso.

```
JT <ReferenciaBool>, Etiqueta
JF <ReferenciaBool>, Etiqueta
```

  JT salta a la etiqueta si la referencia vale "Verdadero", continuando la ejecución en la siguiente instrucción si no es así. JF tiene el efecto contrario, salta si el valor Bool vale "Falso".

  Es útil disponer de una variante que permita eliminar directamente el objeto Bool utilizado. Esto es lo que hacen las instrucciones anteriores, a las que se añade una D (de *Delete*, borrar).

```
JTD <ReferenciaBool>, Etiqueta
JFD <ReferenciaBool>, Etiqueta
```

- **Comprobaciones sobre referencias**. En muchos casos lo que se pretende comprobar es simplemente si un objeto existe o no, es decir, si una referencia está libre o no.

```
JNull <Referencia>, Etiqueta
JNNull <Referencia>, Etiqueta
```

  JNull salta a la etiqueta si la referencia está libre (es *Null*, nula). JNNull salta si la referencia está ocupada (no está libre o no es nula, *Not Null*).

14.3.5.6 Manejo de Excepciones

Instrucción Handler

```
Handler <etiqueta>
```

La misión de la instrucción Handler (manejador) es indicar la dirección donde se debe continuar el flujo de ejecución en caso de que ocurra una excepción (la dirección del manejador). La dirección será aquella que corresponda a la posición de la etiqueta en el código fuente.

Instrucción Throw

```
Throw;
```

Throw (lanzar) lanza una excepción. Cuando se ejecute un Throw, la ejecución del programa continuará en la dirección del último manejador ejecutado. En caso de que no haya ninguno, la ejecución se dará por finalizada. Los distintos manejadores se van apilando de manera que tienen prioridad los últimos que se hayan ejecutado.

14.3.6 Jerarquía de Clases Primitivas

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 que se pueden considerar como definidas en dicho área de manera permanente.

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. A continuación se listan dichas clases básicas:

- **Object**. Clase raíz de la jerarquía.
- **Bool**. Valores booleanos.
- **Integer**. Números enteros.
- **Float**. Números en coma flotante.
- **String**. Cadenas de caracteres.
- **Array**. Vectores unidimensionales de objetos.

Adicionalmente, se incluyen transitoriamente clases para la sincronización y la entrada/salida que posiblemente sean sustituidas por subsistemas de un sistema operativo, y una clase reloj que se introduce en esta versión para controlar tiempos de ejecución y hacer pruebas de rendimiento:

- **Clock**. Cronómetros
- **Semaphore**. Semáforos para sincronización
- **Stream**. Flujos de Entrada/Salida. Tiene dos subclases:
  - **FileStream**. Ficheros
  - **ConStream**. Consola

15 Diseño e Implementación del Prototipo Inicial de la Máquina Abstracta

15.1 Introducción

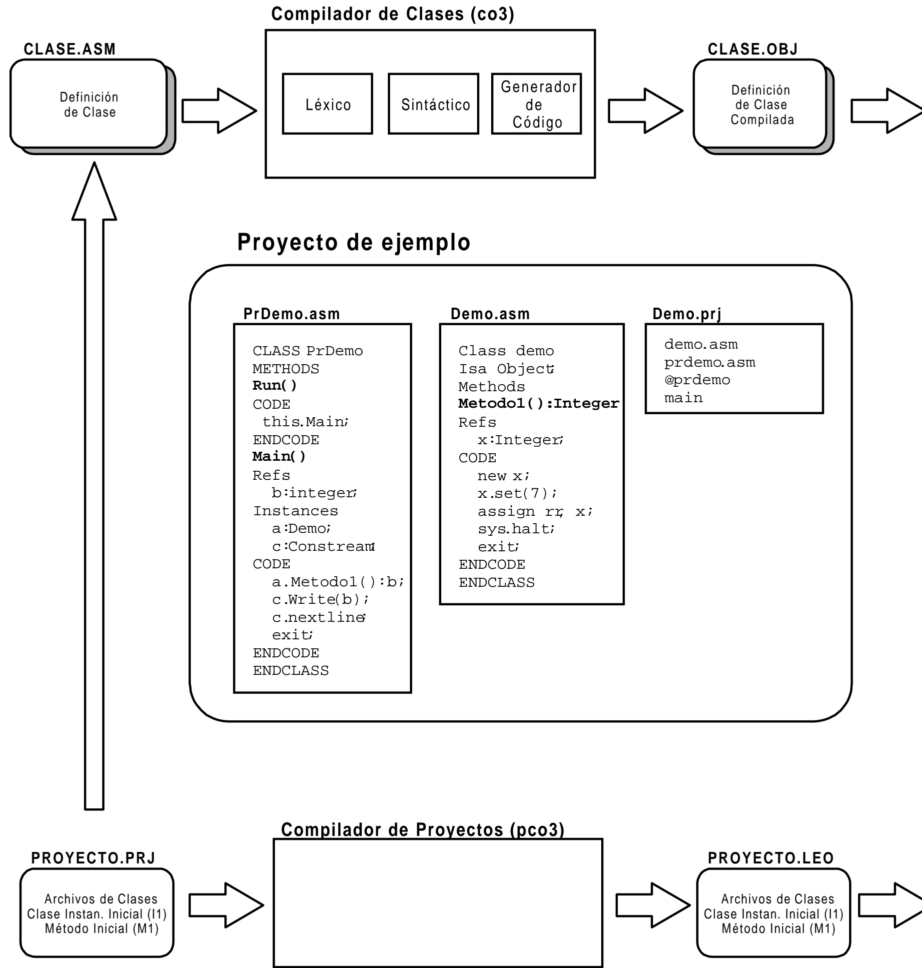
En este apartado se describirá la filosofía de implementación de un prototipo de la máquina abstracta Carbayonia. La implementación de este prototipo fue desarrollada en una primera versión como proyecto fin de carrera de la Escuela Superior de Ingenieros Informáticos de la Universidad de Oviedo. Se pueden consultar los detalles en [Izq96]. Este prototipo fue posteriormente modificado para después incluir una versión del mecanismo de protección propuesto en este trabajo en otro proyecto fin de carrera [Gar99], donde puede encontrarse más información al respecto.

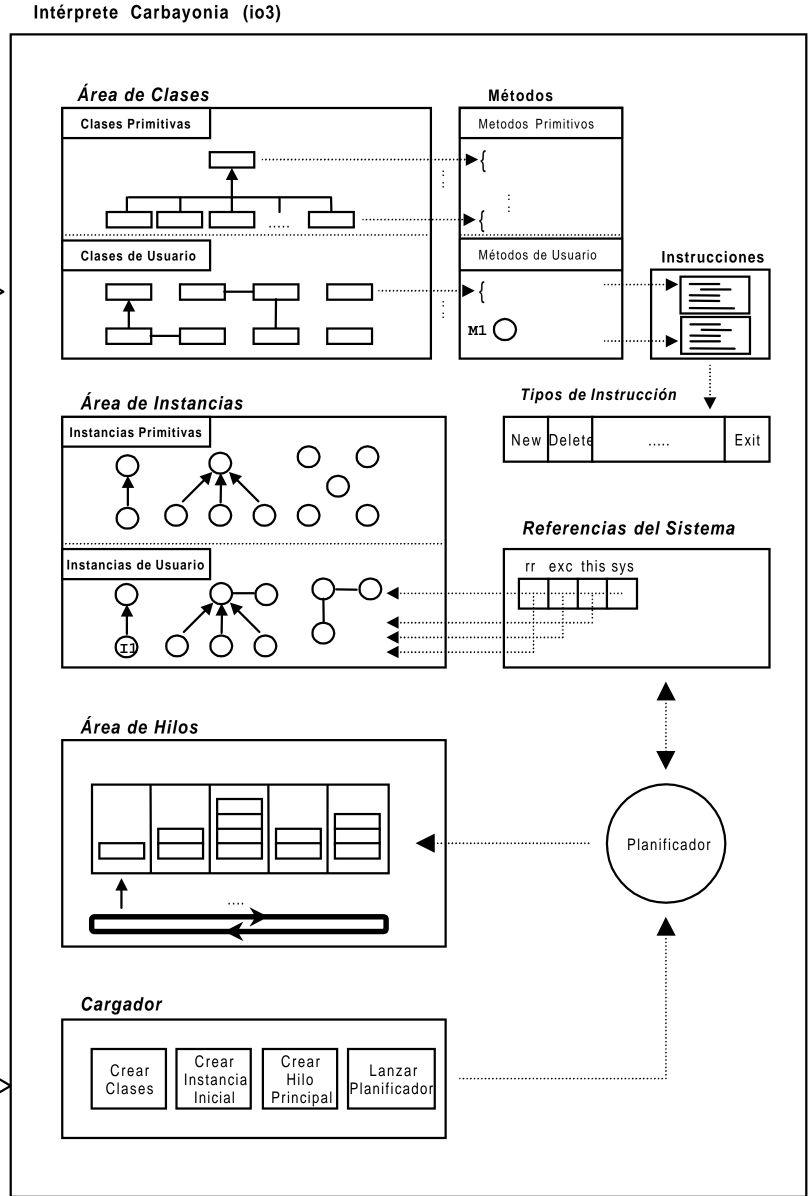
En este apartado se va a dar una descripción de la implementación sin llegar a un gran nivel de detalle. En la documentación de los proyectos anteriores podrá obtenerse una descripción más completa.

En la siguiente figura se muestra un esquema general del funcionamiento y estructura de la máquina.

El código fuente de cada clase se coloca en un fichero con extensión .asm. El conjunto de las clases que forman un proyecto, así como la clase y el método iniciales que se tienen que instanciar para iniciar la ejecución se especifican en un fichero .prj. Un compilador de clases (co3) y uno de proyectos (pco3) generan la información correspondiente en formato binario que entiende ya el intérprete de la máquina Carbayonia (io3). Las clases se almacenan en ficheros .obj y los proyectos en ficheros .leo.

El intérprete se invoca pasando como parámetro el fichero .leo correspondiente a la aplicación que se quiere ejecutar. Éste se pasa a un cargador, que se ocupa de crear las clases especificadas en el proyecto (cargándolas a partir de sus ficheros binarios .obj correspondientes). A continuación se crea la instancia de la clase inicial, se crea un hilo de ejecución principal para invocar el método inicial y se lanza el planificador. A partir de ese momento se simula el funcionamiento normal de la máquina, ejecutando instrucciones del método, etc.


CLASE.ASM
Definición de Clase
Compilador de Clases (co3)
Léxico
Sintáctico
Generador de Código
CLASE.OBJ
Definición de Clase Compilada
Proyecto de ejemplo
PrDemo.asm
CLASS PrDemo
METHODS
Run()
CODE
this.Main;
ENDCODE
Main()
Refs
b:integer;
Instances
a:Demo;
c:Constream
CODE
a.Metodo1():b;
c.Write(b);
c.nextline
exit;
ENDCODE
ENDCLASS
Demo.asm
Class demo
Isa Object;
Methods
Metodo1():Integer
Refs
x:Integer;
CODE
new x;
x.set(7);
assign rr, x;
sys.halt;
exit;
ENDCODE
ENDCLASS
Demo.prj
demo.asm
prdemo.asm
@prdemo
main
PROYECTO.PRJ
Archivos de Clases
Clase Instan. Inicial (I1)
Método Inicial (M1)
Compilador de Proyectos (pco3)
PROYECTO.LEO
Archivos de Clases
Clase Instan. Inicial (I1)
Método Inicial (M1)


Intérprete Carbayonia (io3)
Área de Clases
Clases Primitivas
Clases de Usuario
Métodos
Metodos Primitivos
Métodos de Usuario
M1
Instrucciones
Tipos de Instrucción
New
Delete
.....
Exit
Área de Instancias
Instancias Primitivas
Instancias de Usuario
I1
Referencias del Sistema
rr exc this sys
Área de Hilos
Planificador
Cargador
Crear Clases
Crear Instancia Inicial
Crear Hilo Principal
Lanzar Planificador

15.2 Diseño orientado a objetos

El alto nivel de la interfaz que proporciona la máquina permite la utilización de diferentes diseños para la implementación. El objetivo de este prototipo es el de proporcionar una plataforma de trabajo para experimentar con ella, y realizar cambios, de forma que aunque éstos sean radicales, puedan ser fácilmente soportados. Así pues, el rendimiento no es un objetivo principal.

La idea fundamental de la implementación consiste en reproducir con objetos los elementos de la máquina. Se pretende imitar con objetos del programa todos los elementos de la máquina.

Clases, instancias, referencias, métodos e instrucciones de los métodos serán objetos del programa (TClass, TInstance, TRef, TMethod, TInstruction). Estos objetos se agruparán dentro de contenedores según su tipo: área de clases, área de instancias y área de referencias.

Los objetos tendrán un comportamiento con métodos que realicen las funciones de los elementos de la máquina. Así por ejemplo, los objetos método tendrán una operación invokeMethod que implemente toda la semántica de la operación de invocación de ese método en la máquina.

Las clases concretas especifican las clases abstractas para representar entidades específicas. Por ejemplo, clases primitivas de la máquina heredan de TClass, como TCObject, TCInteger, TCFloat, TCString así como clases no primitivas, del usuario TCUserClass. El mismo proceso se utiliza para TInstances, TMethods y TInstructions. De esta forma, pueden ser añadidas fácilmente nuevas clases primitivas, métodos adicionales y nuevas instrucciones.

Las relaciones entre los elementos de la máquina están representadas por medio de relaciones entre los objetos correspondientes en el programa. Por ejemplo, las clases tienen una lista de objetos método, antecesores, agregados y asociados. Las referencias están compuestas por un nombre, un tipo y el identificador de la instancia a la que apunta, etc.

Para implementar los hilos de ejecución se utiliza un mecanismo similar al de la Máquina-P [NAJ+76] diseñada para el lenguaje Pascal. Cada objeto hilo (TThread) tiene una pila asociada compuesta por contextos. Un contexto representa el contexto de ejecución de un método. Cada vez que en un hilo se llama a un método se apila un contexto. El contexto proporciona los elementos dinámicos de ejecución de un método: las referencias e instancias locales, la instancia sobre la que actúa el método (this), un lugar donde dejar el valor de retorno, etc. Las instrucciones del método actuarán sobre la información del contexto. Al finalizar el método se elimina el contexto.

La máquina tiene incorporado un planificador que va alternando la ejecución de los métodos entre los diferentes hilos, con una política PEPS (Primero en Entrar, Primero en Salir). Sin embargo este soporte es transitorio hasta que sea diseñado un verdadero soporte para la concurrencia a nivel de sistema operativo.

Para implementar las excepciones se hace que la pila de un hilo pueda entremezclar elementos para representar los manejadores de excepciones.

El resultado es una estructura en tiempo de ejecución en la que los objetos del programa reproducen los elementos conceptuales del modelo de objetos y las relaciones que existen entre ellos según indican los programas escritos en el lenguaje Carbayón. Es decir, que básicamente se hace una representación con objetos en tiempo de ejecución de la estructura de los programas Carbayonia. La siguiente figura muestra el diagrama general de clases del prototipo.

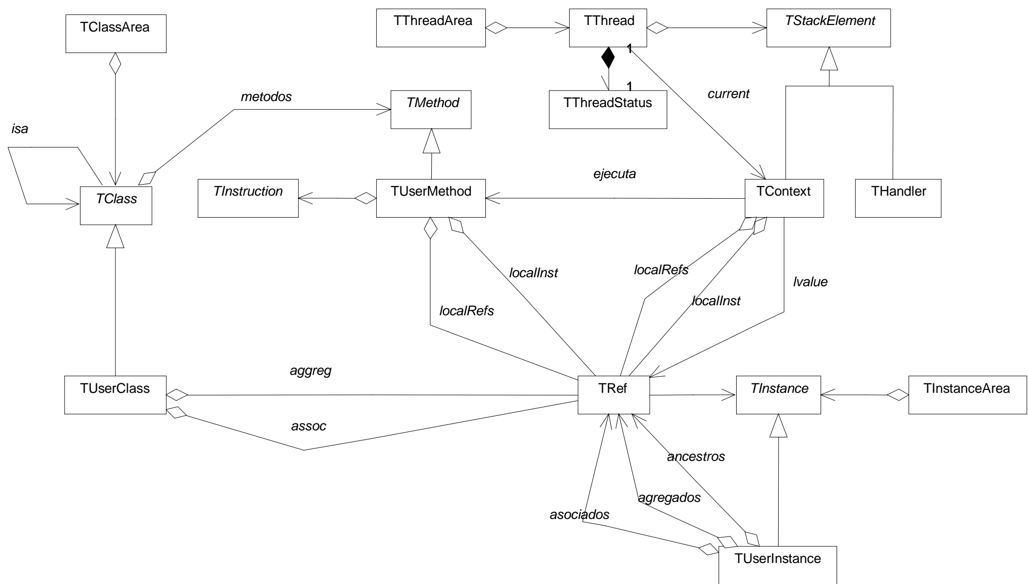


Figura 15.2: Diagrama de clases del intérprete de la máquina

15.3 Implementación primitiva de elementos básicos

En un sistema integral totalmente orientado a objetos, todos los elementos que la componen se describen en términos de objetos. De esta manera las clases se definen en función de otras clases, estas a su vez en función de otras y así sucesivamente. Sin embargo, esta recursión debe finalizarse en algún momento. Deben existir algunas clases que no se definan en función de otras, su definición debe ser proporcionada directamente por la máquina.

15.3.1 Clases primitivas y clases de usuario

La máquina debe proporcionar una implementación primitiva para ciertos elementos, que permita finalizar esa aparente recursión infinita. Estos elementos serán fundamentalmente las clases, con todo lo que ello conlleva: representación de las instancias y sus métodos. A las clases, métodos e instancias que son implementados directamente por la máquina se les denomina **clases, métodos e instancias primitivas**. Al resto, que se definen de manera normal a partir de otras clases se llaman **clases, métodos e instancias de usuario**.

En cualquier caso, el uso de estas clases primitivas no se debe diferenciar del uso de las clases de usuario. Esto quiere decir, que por ejemplo, cuando se invoca un método de una clase primitiva se hace de la misma manera que para una clase de usuario. La diferencia será que la máquina, con la clase primitiva invocará un método primitivo de la misma (implementado internamente en la máquina, en este caso en C++) que realice la función de ese método. Si es una clase de usuario, se invocará un método normal de la máquina, ejecutando las instrucciones de la máquina que describen ese método. Este mecanismo que no diferencia en la interfaz la utilización de los elementos primitivos, accediendo transparentemente a la implementación primitiva de los mismos es similar a la técnica utilizada en la máquina de Smalltalk [GR83].

	Primitivas	de Usuario
Clases	No necesitan ser definidas en términos de otras clases. La máquina les da soporte directo.	Se definen en términos de otras clases (agregados y asociaciones), que les dan soporte.
Métodos	Son implementados directamente por la máquina. No necesitan cuerpo.	Son implementados por instrucciones normales de la máquina. Necesitan el cuerpo con estas instrucciones

Tabla 15.1: Características básicas de las clases en Carbayonia

Como se mencionó anteriormente, la manera de implementar esta activación transparente de los elementos primitivos es mediante el uso de una clase abstracta que representa cada elemento en general. Esta clase se utiliza para la interacción con el resto de los elementos de la máquina. A partir de esa clase se derivan los elementos implementados primitivamente y los elementos de usuario, incorporando en los primitivos una implementación directa de su comportamiento y en los de usuario su representación y comportamiento según define la especificación del sistema.

15.4 Representación de los elementos de la máquina

15.4.1 Representación de las clases y del área de clases

El área de clases almacena todas las clases de la máquina. Al inicializarse el intérprete se registran en esta área todas las clases primitivas definidas en la especificación de la máquina.

En el diseño original no se dispuso de ningún método para realizar la carga y descarga dinámica de clases en tiempo de ejecución. En la presente versión se abrió esta puerta al implementar los métodos LoadClass() y UnLoadClass() de la clase reflectiva System que permite controlar el funcionamiento interno de la máquina.

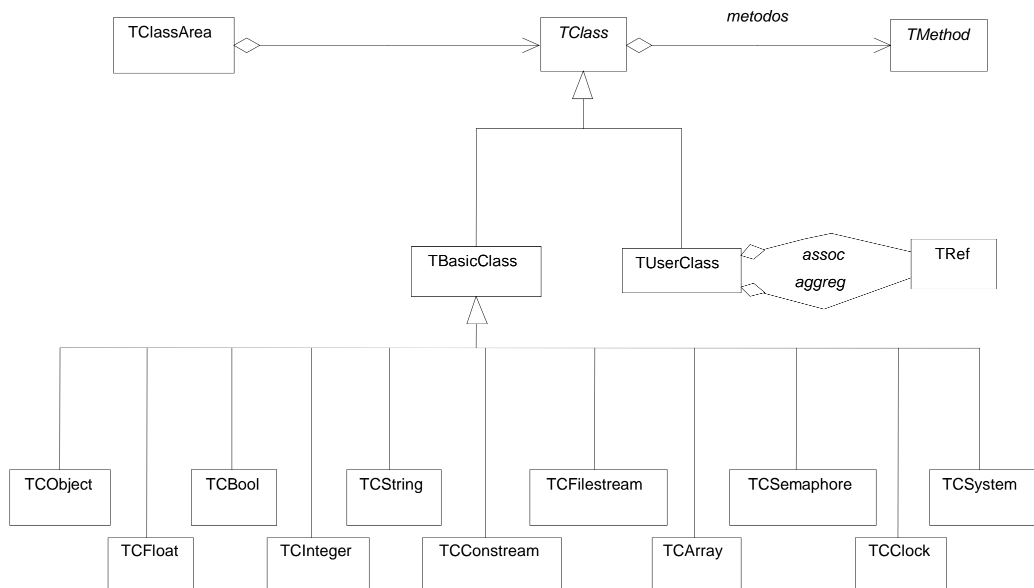


Figura 15.3: Jerarquía de TClass

En la figura 15.3 se muestra el área de clases (TClassArea) y su relación con la clase base TClass, que es el ancestro de todas las clases de la máquina. Un TClass viene definido por el nombre de la clase y está asociado a un conjunto de métodos (TMethod) que representan los métodos de la clase.

De TClass se deriva TBasicClass, de la que derivan a su vez todas las clases primitivas de la máquina: TCObject, TCFloat, TCBool, TCInteger, TCString, TCConStream, TCFileStream, TCArray, TCSemaphore, TCClock (reloj, añadido en esta versión) y TCSystem (clase reflectiva para controlar la propia máquina, añadida en esta versión).

De TClass también deriva TUserClass, que representa las clases de usuario. Una clase de usuario (TUserClass) consta de los siguientes elementos:

- Conjunto de clases **Isa** (Herencia), de **agregaciones** (aggreg, un conjunto de referencias TRef) y de **asociaciones** (assoc)
- Conjunto de **métodos** de usuario (TUserMethod), definidos mediante un **Nombre**, **Parámetros** (un vector de TRef), Clase del valor de retorno (un string), **Referencias** (Refs, un vector de TRef), **Instancias** (Instances, un vector de TRef).

15.4.2 Representación de los métodos

La clase abstracta TMethod es la representación abstracta de un método perteneciente a una clase. De ésta se derivan TPrimitiveMethod (para definir los métodos de las clases primitivas, existirá uno por cada método) y TUserMethod (para los métodos de las clases de usuario).

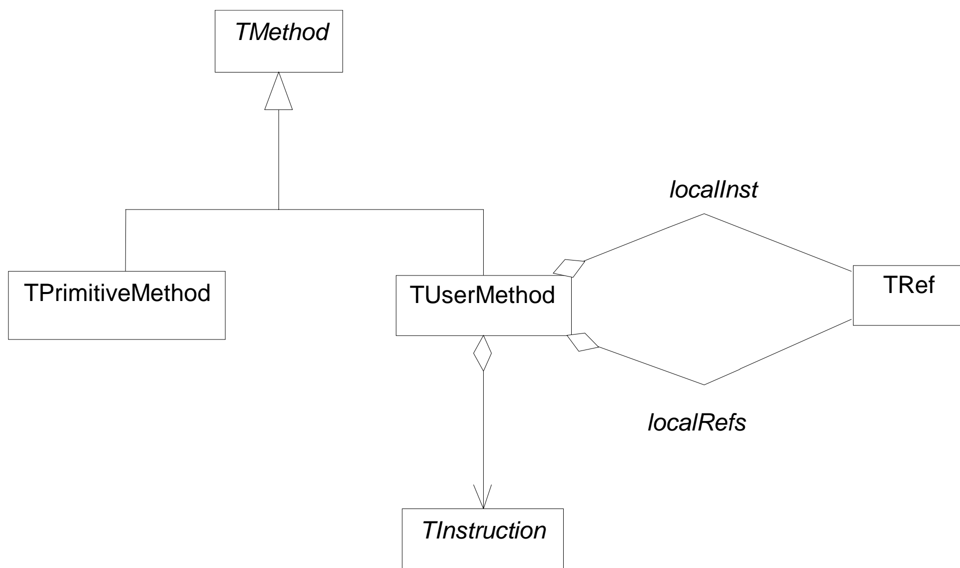


Figura 15.4: Representación de la clase TMethod

En esta nueva versión se han ampliado los métodos de TMethod y sus descendientes para conocer en tiempo de ejecución el número de parámetros del método y su tipo y el tipo de retorno. Éstos se han implementado como nuevos métodos de la clase primitiva Object.

15.4.3 Representación de las instancias y del área de instancias

En la figura 15.5 se representa el área de Instancias (TInstanceArea). Este área almacena las instancias en tiempo de ejecución de las clases definidas en el área de clases. Cada instancia presente en el área se identifica con un número único. Este valor puede accederse en tiempo de ejecución mediante el método primitivo de la clase Object GetID().

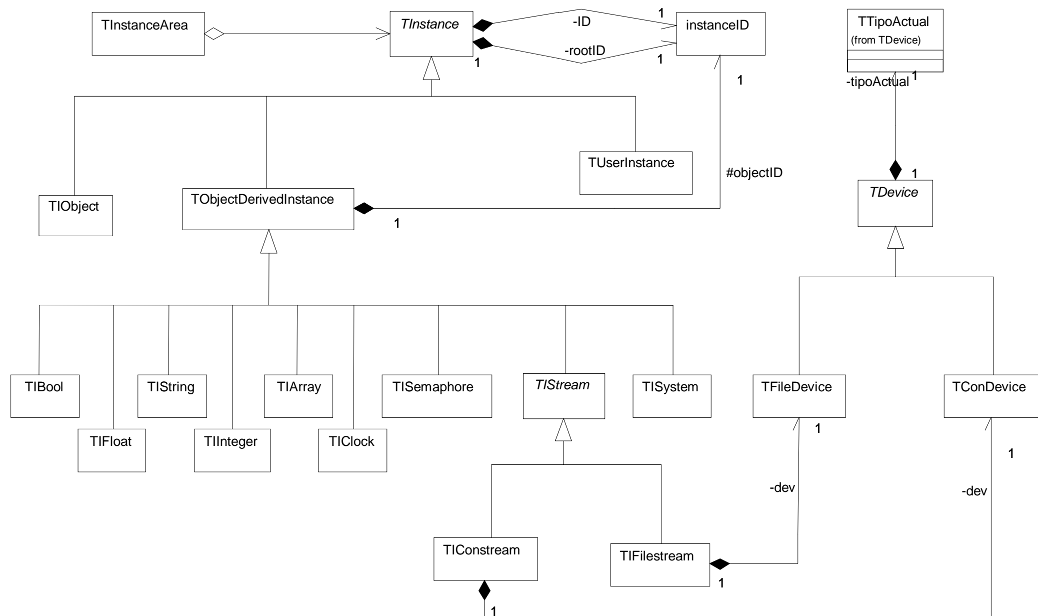


Figura 15.5: Diagrama de clases del área de instancias

El área de instancias se representa como una colección de objetos Tinstance, que representa una instancia individual en el sistema. De TInstance se deriva la subclase TobjectDerivedInstance, que es la base de todas las instancias de clases primitivas (TIBool, TIFloat, TIString, etc.). Cada una de sus subclases implementa un atributo valor para contener el valor de la instancia (por ejemplo: int valor, para TInteger).

La otra clase que deriva de TInstance es TuserInstance, que representa las instancias de usuario en tiempo de ejecución. Al igual que TUserClass, contiene una conjunto de referencias para sus ancestros (instancias que se crean automáticamente al procesar el área Isa de la clase), sus agregaciones (agg) y asociaciones (ass).

15.4.4 Representación de las referencias

La clase TRef representa las referencias, y tiene la función de asociar un nombre simbólico con un nombre de clase y un identificador de instancia del área de instancias.

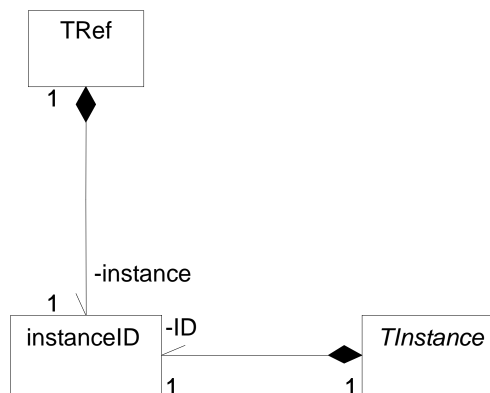


Figura 15.6. Diagrama de clases de Referencias

El área de referencias no está implementada como tal, si no que sólo se define la clase TRef para representar las referencias. Fácilmente podría haberse creado una clase TRefArea para explicitar la agrupación de las referencias en un contenedor común, lo que facilita posteriormente tareas como la recolección de basura, a costa posiblemente de algo de pérdida

de rendimiento. No se implementó aún por razones de compatibilidad con el diseño anterior y por no afectar directamente a la parte de introducción del mecanismo de protección.

15.5 Funcionamiento general de la máquina

Una vez descritas las áreas principales del intérprete, en esta sección se van a describir, mediante diagramas de colaboración, las partes más relevantes del funcionamiento de la máquina, intentando con ello, que el lector se haga una idea general de su modo de trabajo.

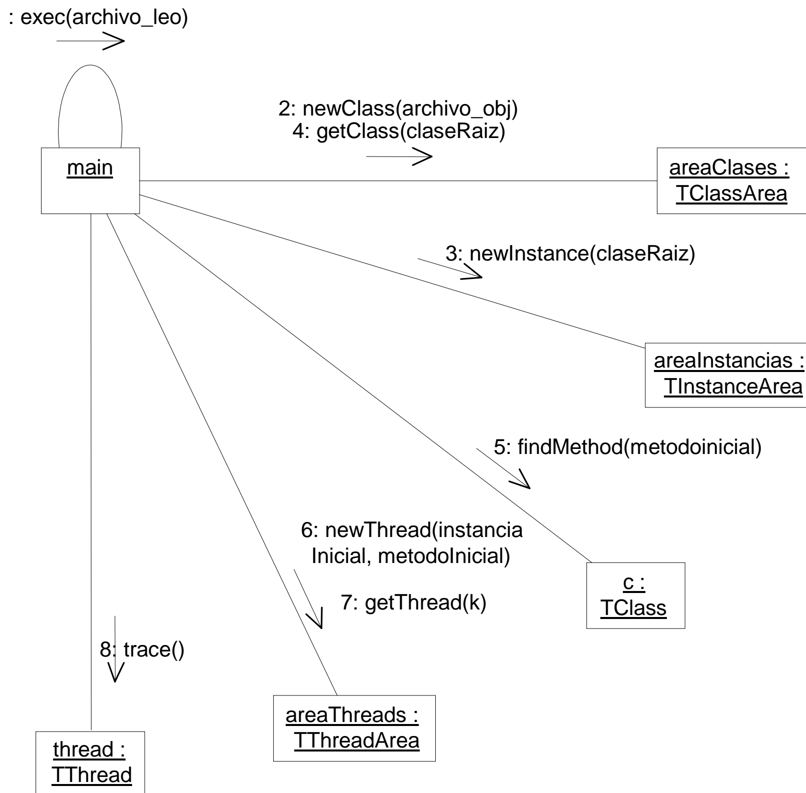


Figura 15.7: Esquema general de funcionamiento del intérprete

Desde la línea de órdenes se invoca al intérprete (máquina) con un fichero LEO como parámetro. La primera operación que realiza es procesar los archivos de entrada, invocando al método newClass del área de clases para cada archivo de definición de clase (OBJ) que se encuentre en el archivo LEO de entrada (2). A continuación se leen del archivo LEO la clase Raíz y el método inicial para ejecutar la aplicación, creando una instancia de la clase (3) y localizando el método inicial (4,5). Por último, se crea un hilo con la instancia y el método inicial (6).

Una vez creadas todas las estructuras, se inicia el bucle principal del intérprete que itera sobre todos los hilos del área, ejecutando una instrucción de cada hilo (7,8) mientras haya hilos de ejecución.

15.6 Creación de elementos

15.6.1 Creación de una clase de usuario

En el siguiente diagrama se muestra el proceso de introducción de una nueva clase en el área de clases. El método newClass es invocado por el programa principal al realizar la carga del programa pasado como argumento.

En una evolución del prototipo de la máquina, se implementa la clase reflectiva System que tiene un método llamado LoadClass() que realiza la misma labor.

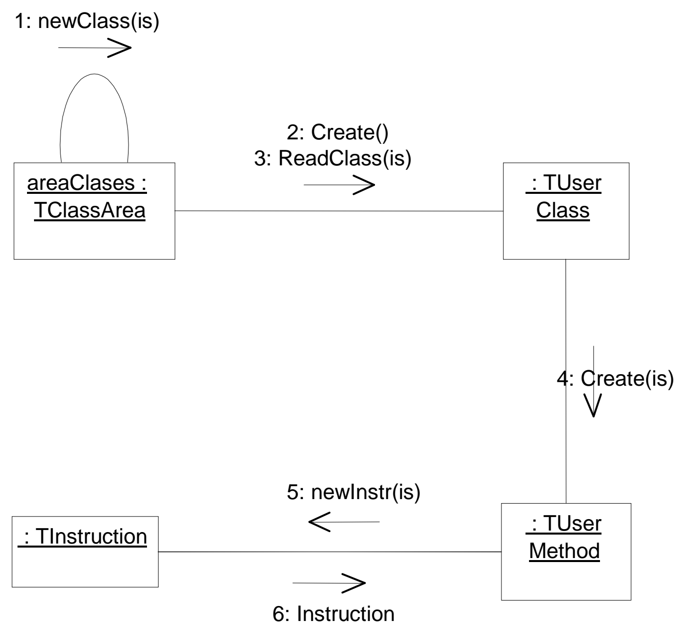


Figura 15.8: Creación de una clase de usuario

Como parámetro se pasa un flujo (is) del que se leerá la definición de la clase. Ésta crea una instancia de TUserClass que será encargada de leer la clase del flujo mediante readClass. Durante del proceso de creación de la clase se generan varios TUserMethod para contener cada uno de los métodos de la clase. A su vez, éstos invocan al método estático de TInstruction, newInstr, para que cree una instrucción del tipo adecuado y añadirla a la lista de instrucciones del método.

15.6.2 Creación de una instancia de usuario

En la siguiente figura se muestra el diagrama de colaboración que representa el proceso de creación de una instancia de usuario, que es un caso más general que la creación de una instancia de una clase primitiva.

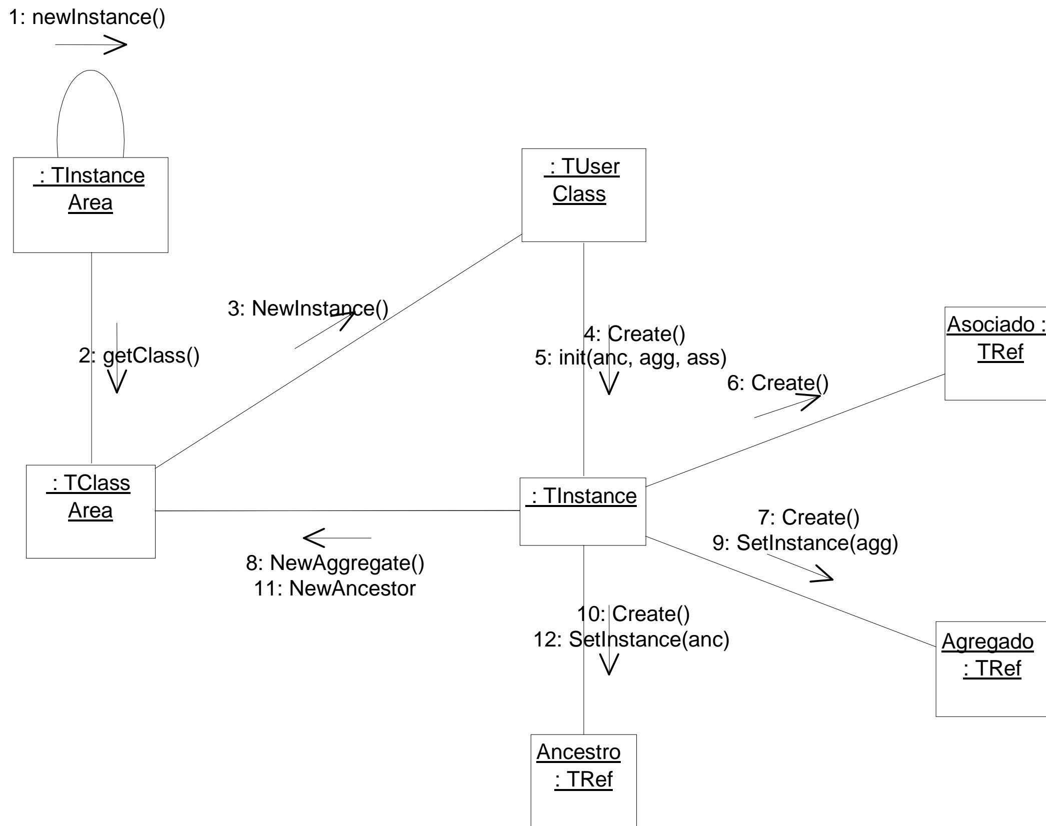


Figura 15.9: Creación de una instancia de usuario

El proceso comienza con la ejecución del método newInstance del área de instancia. En primer lugar se localiza la clase de la que se quiere crear la instancia (2). Si la instancia fuese de una clase primitiva el proceso terminaría al invocar el método newInstance. Sin embargo, al ser una clase de usuario, la clase obtenida será TUserClass y ejecutará un proceso más complejo.

En primer lugar crea una instancia de usuario, (4) y se inicializa con los parámetros: agregados, asociados y ancestros. Con esta información, TUserInstance hace las siguientes operaciones:

- Crea una referencia para cada agregado pasado como parámetro y construye una instancia del tipo correspondiente (por lo que volveríamos al paso 1 en el caso de instancias agregadas de usuario) (6,7,8)
- Crea una referencia para cada asociado
- Crea una instancia ancestro para cada elemento del área Isa de la clase, invocando al método newAncestor del área de instancias (que crea una instancia cuyo padre es la instancia principal).

15.7 Ejecución de instrucciones

15.7.1 Ejecución de una instrucción

En el siguiente escenario se describe el proceso de ejecución de una instrucción.

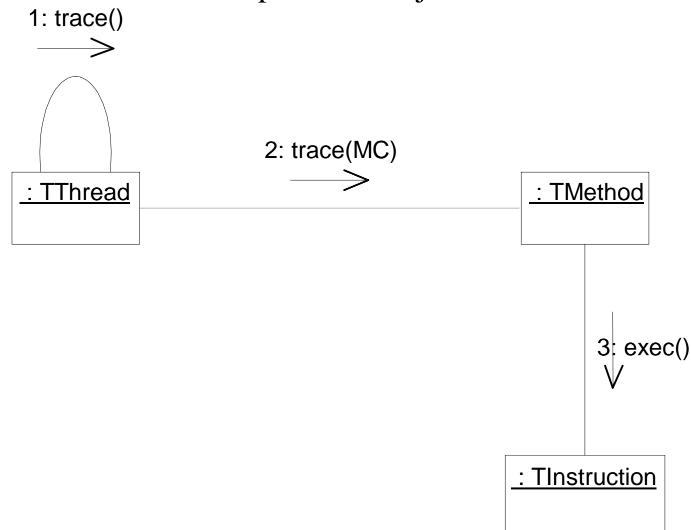


Figura 15.10: Ejecución de una instrucción

El proceso comienza con la invocación del método trace() del hilo. En primer lugar se invoca al método trace del método del contexto actual pasándole como parámetro el contador de método (MC). Este a su vez invoca al método exec del método solicitado. En este punto, dado que cada instrucción implementa su propio exec, se producirá la acción asociada a la instrucción.

En los siguientes apartados se estudiará en detalle la implementación del método exec de las principales instrucciones que conllevan aspectos relacionados con la protección.

15.7.2 Instrucción New <ref>

En el siguiente diagrama se muestra el proceso de creación de una instancia.

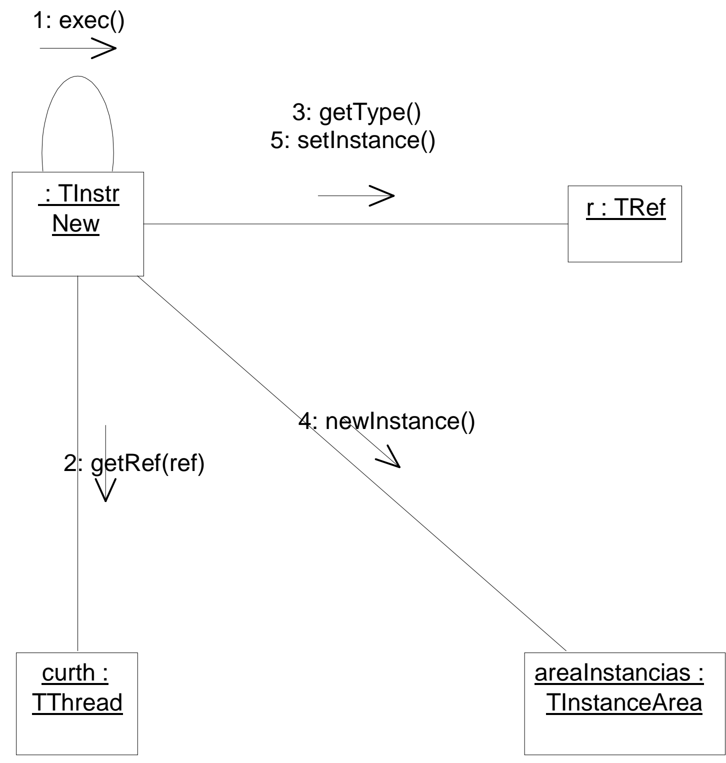


Figura 15.11: Ejecución de una instrucción

Este proceso comienza al invocar TMethod al método exec de la instrucción (véase diagrama anterior). En primer lugar obtiene del hilo la referencia real asociada al parámetro de la instrucción (2). A continuación se obtiene el tipo de la referencia y se crea una instancia en el área de clases del tipo correspondiente (3, 4).

15.7.3 Instrucción Delete

En el siguiente diagrama se muestra la ejecución de la instrucción Delete.

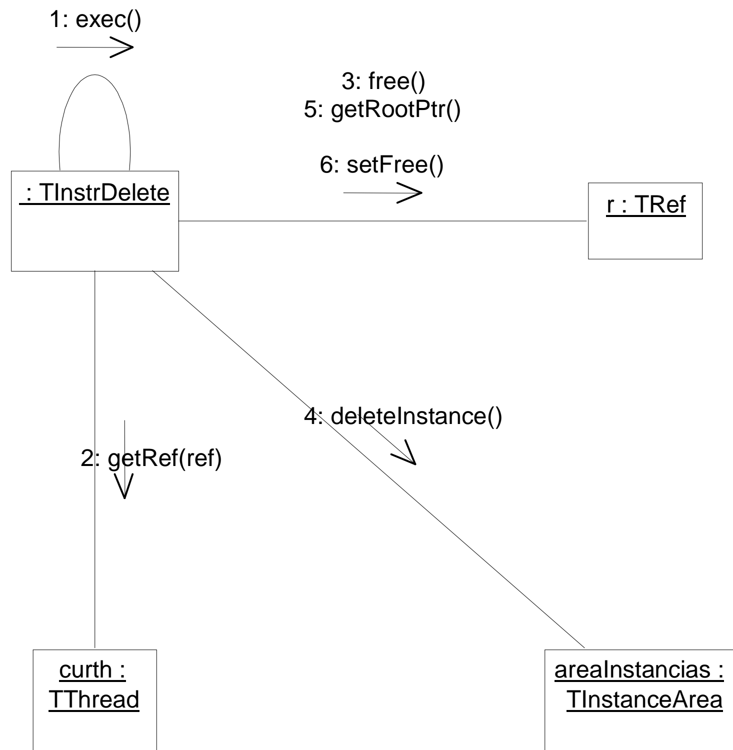


Figura 15.12: Ejecución de la instrucción Delete

En primer lugar se obtiene la referencia real asociada al parámetro y se comprueba que no está libre (3), si lo está se lanza una excepción. A continuación se verifica que no tratamos de liberar una instancia agregada (5) y se procede a liberar la instancia (4). Finalmente, se pasa el estado de la referencia a libre (6).

15.7.4 Instrucción Assign <ref-dest>, <ref-orig>

En el siguiente diagrama se muestra la ejecución de la instrucción Assign.

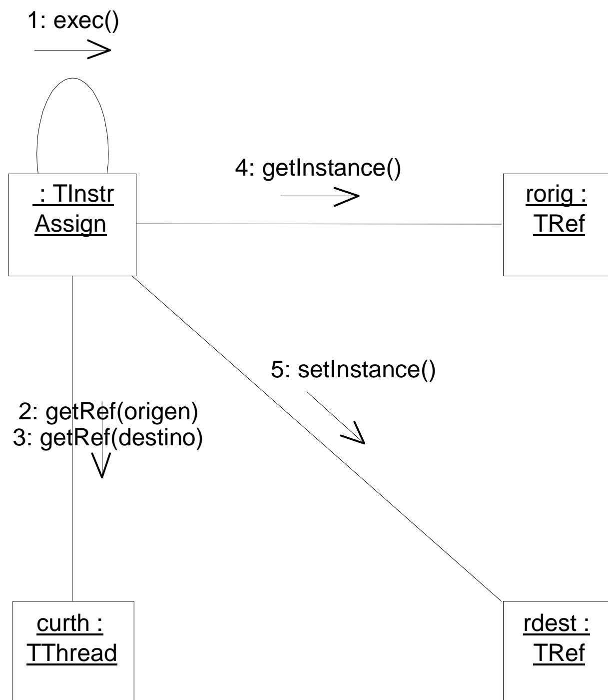


Figura 15.13: Ejecución de la instrucción Assign

Esta instrucción obtiene en primer lugar las referencias asociadas a los parámetros de la instrucción (2, 3) y a continuación procede a copiar la instancia de la referencia de origen sobre la referencia destino (4, 5).

15.7.5 Instrucción Call: <ref>.<método>([<params>])

En este diagrama se muestra el proceso de invocación de un método.

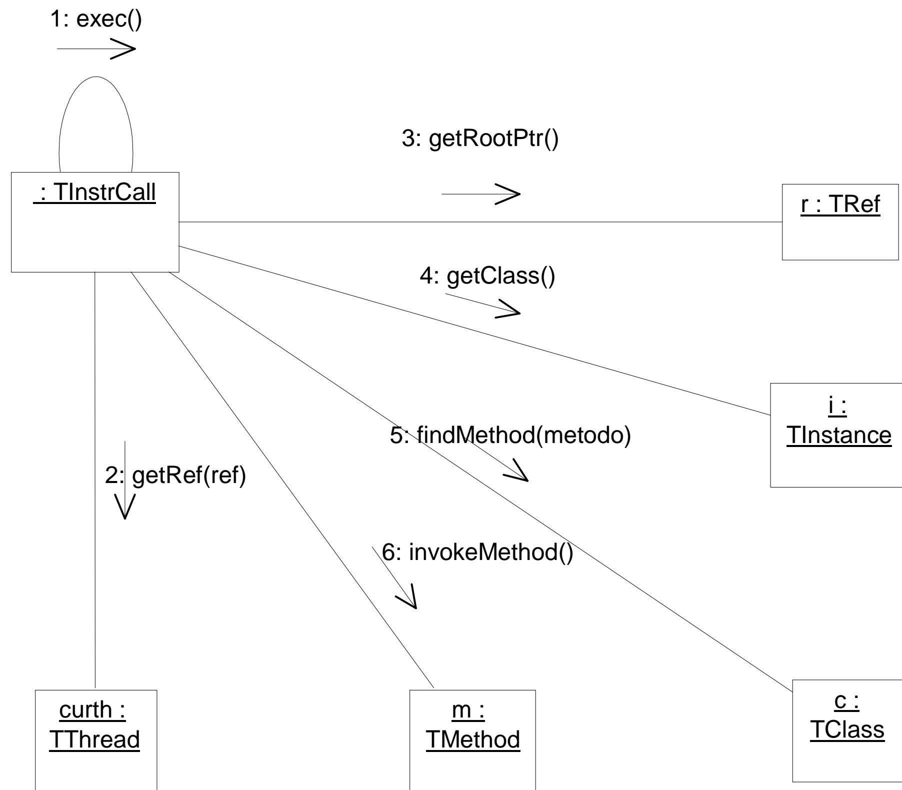


Figura 15.14: Ejecución de la instrucción Call

En primer lugar se busca la referencia a la que se 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 esta nueva implementación del prototipo de la máquina el método se busca en la clase de la instancia a la que apunta la referencia para permitir un uso más genérico de referencias.

16 Implantación del Mecanismo de Protección en la Máquina Abstracta del Sistema Integral Orientado a Objetos

En este capítulo se va a describir el modo en que se ha implantado en un sistema concreto como es Oviedo3 el diseño del núcleo de seguridad descrito en capítulos anteriores. En esencia, las dos decisiones básicas de diseño planteadas fueron las siguientes: uso de capacidades como modelo básico de protección e implantación de éstas al nivel de la máquina abstracta Carbayonia.

16.1 Integración de Capacidades con referencias

Para incluir las capacidades en la máquina, se decidió fusionarlas con las referencias, añadiendo a éstas los permisos sobre los métodos. La siguiente figura representa la nueva estructura de la clase TRef, que representa a las referencias.

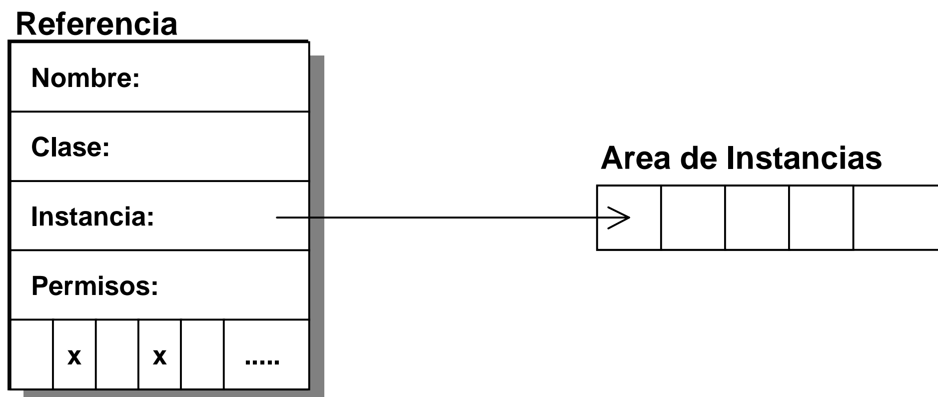


Figura 16.1: Nueva implementación de Referencias para dar soporte a la seguridad

16.1.1 Permisos de longitud variable

Otra decisión de diseño adoptada supuso la creación de permisos de longitud variable, en la que existe un permiso para cada uno de los métodos del objeto al que apunta la referencia, de forma que todos ellos pudieran ser protegidos. Para introducir los permisos en las referencias, se crea una nueva clase, denominada TBitVect, que constituye un vector de permisos. La siguiente figura muestra el nuevo diagrama de clases para esta parte de la máquina.

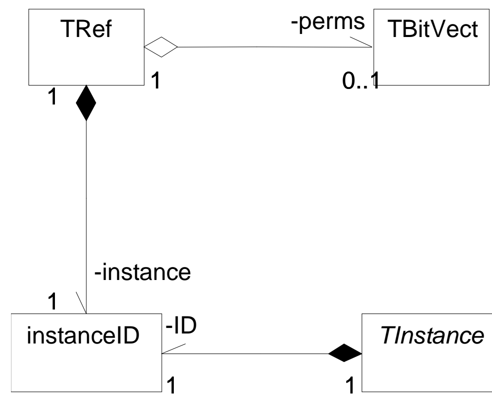


Figura 16.2: Nuevo Diagrama de clases de las referencias

Es importante destacar, en el diseño, que este vector de permisos no se instancia estáticamente cuando aparece la referencia, sino que es creado dinámicamente cuando sea necesario hacer uso del mecanismo de protección.

16.1.2 Permiso de salto de protección

En el diseño del sistema, se tomó la decisión de contar con un permiso especial denominado "Salto de protección", que en su estado activo indica que no es necesario comprobar permisos de ningún método que se invoque porque existen permisos para todos. Así, por ejemplo, cuando se crea un objeto, se asignan todos los permisos al propietario. Bastaría por tanto que el "Salto de protección" estuviese activo.

Conceptualmente puede verse como una abreviatura para indicar el caso en que todos los permisos están activos. Sin embargo, como se verá a continuación, se puede representar especialmente este caso en la implementación (de manera transparente) para mejorar la eficiencia. La manera en que se lleva esta tarea en esta implementación es la siguiente:

Cuando se crea un objeto, el creador obtiene una referencia, que no contará aún con un vector de permisos para los métodos. El modo de implementar este concepto lógico de salto de protección es a través de un método de la clase referencia que comprueba si existe o no el vector de permisos (skipProtection). Es decir, nos indica si el salto de protección está activo o no.

El vector de permisos se instanciará dinámicamente en el momento en que se realice la operación especial de restringir permisos (que se llamará ForbidExec), que sirve para eliminar el permiso sobre un método del objeto al que apunta la referencia. A partir de entonces la referencia cuenta con permisos para todos los métodos salvo para el protegido.

La siguiente figura contiene el código fuente para la clase TBitVect, que implementa el vector de permisos mediante un vector de bits[1].

```
Class TbitVect {
 long *vec;
 int  size;
public:
 TbitVect(int _size) {
       size=_size;
       vec = new long[(_size>>5) + 1];
       for (int i = 0; i< (_size>>5) + 1; i++) vec[i]=~0;
 }
        TbitVect(const TBitVect &v) {
                size = v.size;
                vec = new long[(v.size>>5) + 1];
                for (register int i=0; i < (v.size>>5) + 1; i++)
                        vec[i]=v.vec[i];
        };

 ~TBitVect() {delete vec;}
 inline int  count() { return size;}
 inline bool GetPerm(int i) {return vec[i>>5] & (1 << (i % 32)); }
 inline void SetPerm(int i) {vec[i>>5] |=  1 << (i % 32); }
 inline void ClearPerm(int i) {vec[i>>5] &=  ~(1 << (i % 32)); }
```

Figura16.3: Código fuente de la clase TBitVect

16.2 Implementación de la instrucción de restricción de permisos: ForbidExec

La inclusión del mecanismo de protección aquí desarrollado presenta la ventaja de su fácil utilización a nivel del usuario programador, puesto que se integra totalmente en el modelo de objetos del sistema y el usuario maneja capacidades directamente aunque sin ningún esfuerzo adicional, puesto que van fusionadas a las referencias.

Para una mayor potencia del mecanismo, y que el programador pueda participar en las decisiones de seguridad, tan sólo es necesario añadir una nueva instrucción a la máquina que maneje capacidades, denominada ForbidExec (prohibir o restringir la ejecución). Para ello basta con crear una nueva clase TInstrForbidExec, que hereda de TInstruction (mostrada en negrita en la siguiente figura). Su sintaxis es la siguiente:

```
ForbidExec <referencia>, <método>
```

[1] Se describe aquí la versión ya optimizada de este vector de permisos. Una versión anterior utilizaba un simple vector de valores booleanos.

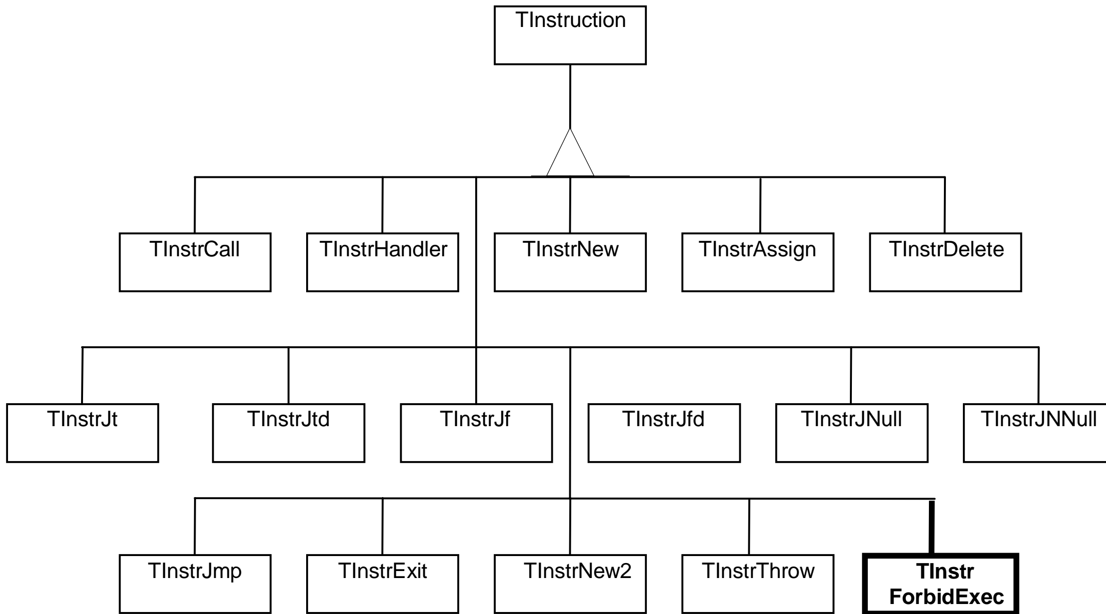


Figura 16.4: Jerarquía de Instrucciones

En el siguiente diagrama se muestra el comportamiento de la máquina frente a esta instrucción.

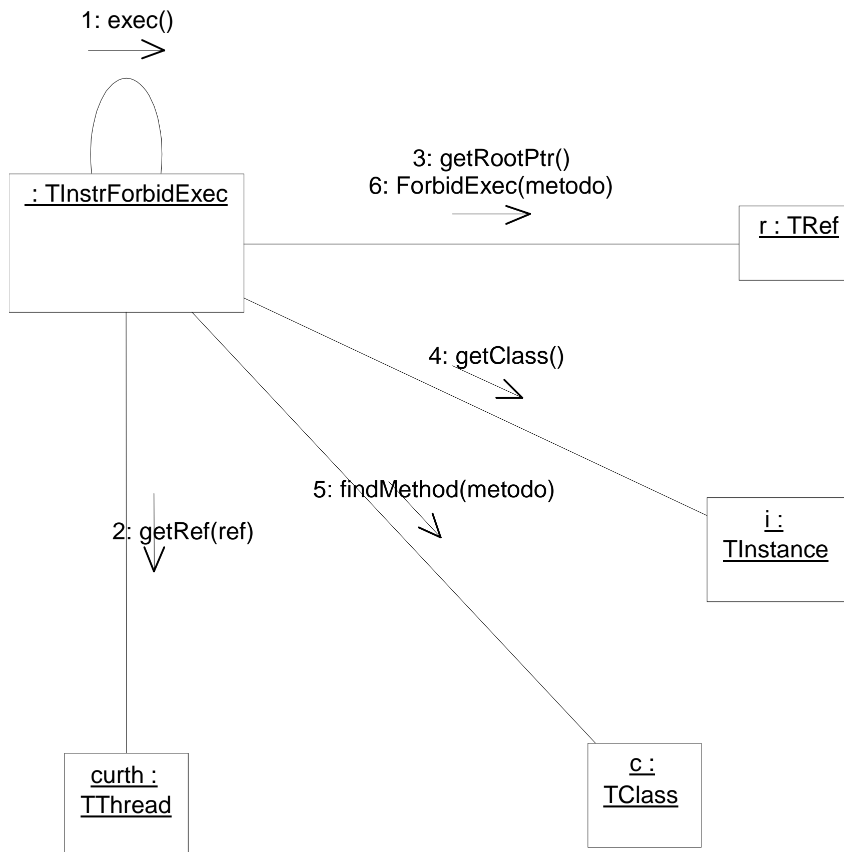


Figura 16.5: Ejecución de la instrucción ForbidExec

En primer lugar se invoca a getRef para obtener la referencia asociada al parámetro 1 de la instrucción (2). A continuación se obtiene la instancia asociada a la referencia (3), se obtiene su clase (4) y a partir de ésta, el *ID* del método indicado como segundo parámetro de la

instrucción. Por último se invoca al método ForbidExec de la referencia para prohibir la ejecución del método a partir de esta referencia. Si es la primera vez que se ejecuta la instrucción se creará el vector de permisos a partir de la clase de la instancia a la que apunta.

Cuando se ejecuta la instrucción ForbidExec, si la referencia no tenía aún vector de permisos, se crea éste con todos los permisos activos salvo el que se acaba de prohibir. Si la referencia ya contaba con permisos, simplemente se desactiva el correspondiente al método que aparece en la instrucción.

Protección de la operación de eliminación

Además del comportamiento anteriormente citado, cuando se protege una referencia con ForbidExec se prohibe su eliminación, lanzando una excepción de seguridad. Así se asegura la protección de la instancia a la que apunta la referencia, impidiendo que aquellos que reciban la referencia invoque la instrucción Delete y destruyan la instancia. Otra alternativa es diseñar la operación de eliminación no como una instrucción, si no como una operación básica de la clase raíz Object, que automáticamente estaría disponible para todos los objetos a través de las referencias. De esta manera la protección de la operación de eliminación se realizaría exactamente igual que el resto de las operaciones de los objetos, mediante un permiso normal.

ForbidExec como un método más de los objetos

Al igual que para la operación de eliminación, que actualmente se presenta bajo la forma de una instrucción de la máquina, se podría hacer también la restricción de permisos de este modo. En lugar de incluir la instrucción adicional descrita anteriormente, se puede añadir una operación a la clase básica Object que desarrolle exactamente la misma funcionalidad de esta instrucción.

En cualquier caso, lo único que cambia es la forma de acceder a la funcionalidad desde los objetos de usuario, no la funcionalidad en sí y el comportamiento del sistema, que en cualquiera de los casos es el mismo.

16.2.1 Creación del vector de permisos

Uno de los problemas que aparecen a la hora de implantar la creación del vector de permisos, es que éste debe ser de tamaño variable, en función del número de métodos del objeto. Su codificación requiere un algoritmo de numeración de métodos de las clases, para establecer la correlación entre el permiso en el vector y el método al que corresponde.

La solución adoptada consistió en crear dos nuevos métodos en TClass: CountMethods() y GetMethodIndex(). Estos dos métodos servirán para conocer el número de métodos de la clase y obtener un valor entero asociado al nombre del método pasado como parámetro. Estos métodos son virtuales en TClass para dar a cada subclase la posibilidad de implementar sus propias versiones de la numeración.

La implementación del método GetMethodIndex() se hace más compleja en la clase del usuario (TUserClass). Las clases de usuario pueden heredar características de un número indeterminado de clases, y por tanto, hay que numerar los métodos de todas las clases de las que deriva la clase.

16.3 Comprobación de permisos: modificación de la invocación de métodos (Instrucción Call)

Con la inclusión de la instrucción ForbidExec y el método TRef correspondiente, se consigue incluir en la máquina el mecanismo básico de protección, pero aún no se ha modificado el comportamiento de la máquina para trabajar con referencias protegidas.

El diseño del mecanismo básico de protección propone modificar el comportamiento de la invocación a métodos para añadir, previamente a la invocación propiamente dicha, un mecanismo de comprobación de protección.

Cuando desde el lenguaje Carbayonia se invoca a un método de un objeto, eso se traduce en la implementación primitiva en C++ en la ejecución del método Exec() de la instrucción Call (por comodidad en lenguaje Carbayón la invocación de un método se realiza concatenando la referencia al objeto con un punto y el nombre del método, pero internamente es equivalente a la instrucción Call).

```
Call <ref>.<método>([<params>])
```

La instrucción ahora debe invocar al método sólo si la referencia posee un permiso de acceso para el método o bien está activo el salto de protección. El siguiente diagrama de interacción de objetos muestra los pasos envueltos en la invocación de un método. Este diagrama pertenece a la implementación de la operación Exec() de una instrucción Call (clase TinstrCall). La ejecución de cualquier instrucción de la máquina se hace a través de la invocación del método Exec() de dicha instrucción.

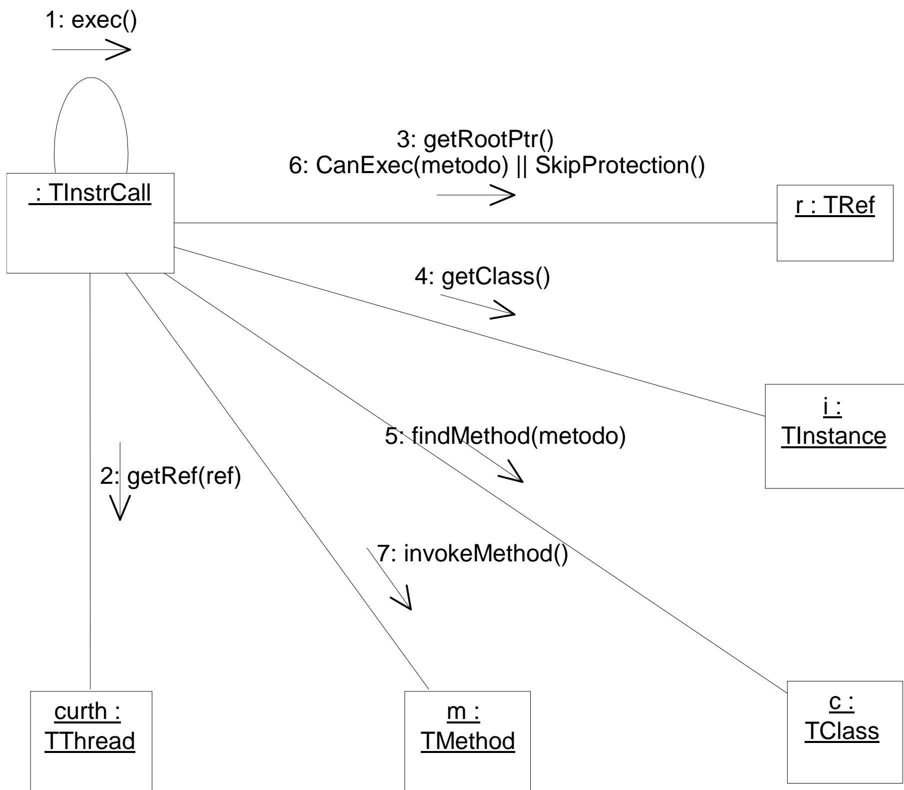


Figura 16.6: Ejecución de la instrucción Call

En primer lugar se busca la referencia a la que se 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. En este punto se verifican los permisos de ejecución de la

instrucción o bien que no sea necesario comprobarlos (salto de protección) (6) y en caso de no ser posible su ejecución se lanza la excepción TSecurityException. Este paso es el paso adicional que debe incluirse para dar soporte al mecanismo de protección. Si la invocación es posible ya se puede invocar el método en cuestión (7).

La instrucción Call utiliza para verificar la protección los métodos SkipProtection() y CanExec() de la referencia cuyo método se desea invocar.

El método SkipProtection() verifica si la referencia tienen vector de permisos y en caso de no tenerlos se procede a invocar al método directamente, puesto que esto significa que están todos activos. Esto es más eficiente, al evitar búsquedas y comparaciones de permisos sobre los métodos.

El método CanExec() es invocado por la instrucción Call si el método anterior devolvió falso. Esto quiere decir que no se tienen todos los permisos y por tanto hay que comprobar si se puede ejecutar el método solicitado. Su labor consiste en buscar el identificador del método con GetMethodIndex() y buscar en el vector de bits su permiso de ejecución (el índice del bit en el vector representa el permiso sobre el método con ese índice).

A continuación se muestra el fragmento de código modificado en la instrucción Call para implantar la comprobación de permisos:

```
[...]
if (r.SkipProtection() ||  r.CanExec(metodo))//Evalua en cortoc.
       method->invokeMethod(th,
r.getInstancePtr()->dinamicCast(method->getClass()),params,lvalue);
else
       throw TSecurity;  // No se puede ejecutar el método.
[...]
```

16.4 Modificación de la instrucción Assign

La máquina cuenta con una instrucción de asignación de referencias. Si se tiene una referencia que apunta a un objeto se puede hacer que otra referencia apunte al mismo. Esto se hace mediante la instrucción Assign. Su sintaxis es la siguiente:

```
Assign <ReferenciaDestino>, <ReferenciaFuente>
```

La implementación de la instrucción Assign antes de incluir la seguridad, sólo copia el identificador de la instancia (instanceID) de la referencia origen sobre la referencia destino. Con la inclusión de protección la modificación que requiere es sencilla: se han de copiar también los permisos. Para ello, se hace uso de un nuevo método de TRef, ClonePerms(), que duplica el vector de permisos sobre la referencia pasada como parámetro.

En el siguiente diagrama se muestra la ejecución de la instrucción Assign.

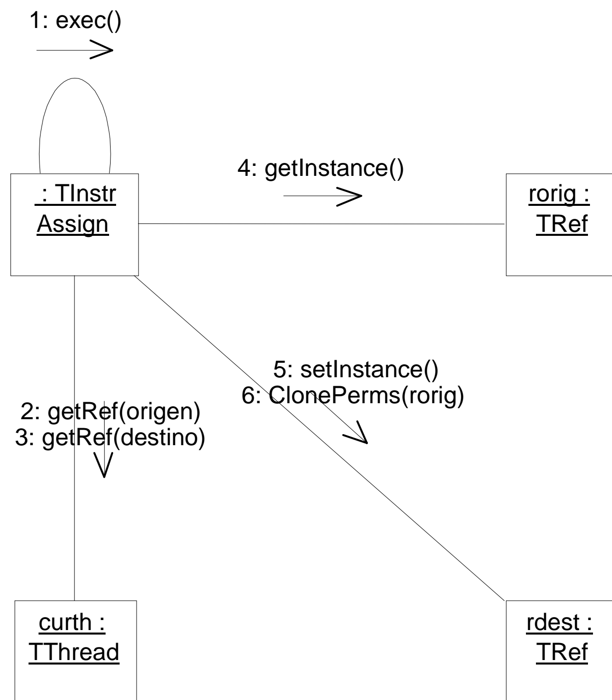


Figura 16.7: Ejecución de la instrucción Assign

Esta instrucción obtiene en primer lugar las referencias asociadas a los parámetros de la instrucción (2, 3) y a continuación procede a copiar la instancia de la referencia de origen sobre la referencia destino (4, 5) y copiar sus permisos invocando al método ClonePerms() (6).

16.5 Modificación de la instrucción New

La instrucción New, que crea un nuevo objeto de una clase a través de una referencia, también necesita una pequeña adaptación. Al crear un objeto ahora hay que devolver los permisos iniciales para ese objeto. Como es para el creador del objeto, se devuelven todos los permisos, lo cual se representa mediante el "salto de protección". Para ello simplemente se añade una llamada a ClearSecurity() sobre la referencia pasada como parámetro, que destruye el vector de permisos que pudiera tener antes la referencia (lo cual es equivalente a concederlos todos). Esto se hace así para asegurar el salto de protección en caso de que la referencia apuntase previamente a una instancia con permisos.

En el siguiente diagrama se muestra el proceso de creación de una instancia.

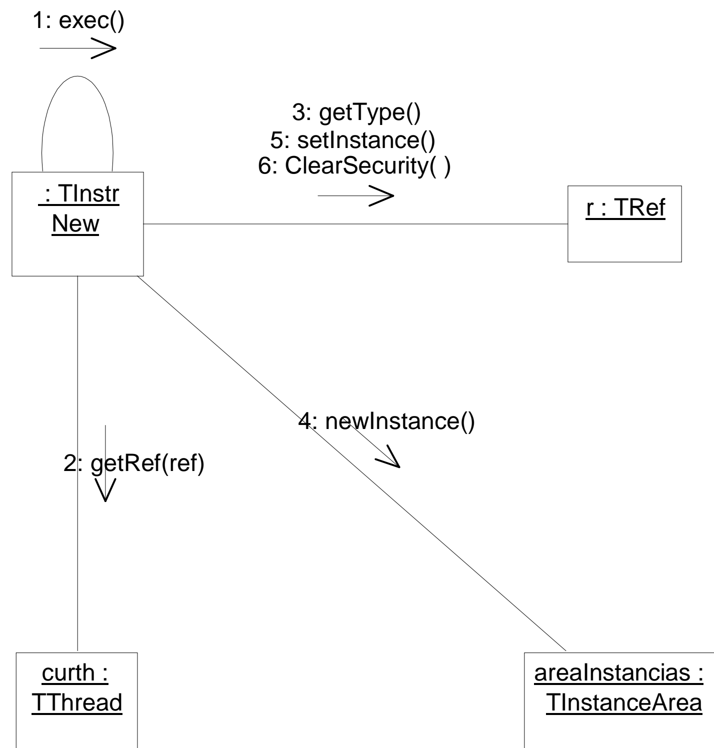


Figura 16.8: Ejecución de la instrucción New

En primer lugar obtiene del hilo la referencia real asociada al parámetro de la instrucción (2). A continuación se obtiene el tipo de la referencia y se crea una instancia en el área de clases del tipo correspondiente (3, 4).

Por último, se asocia la nueva instancia a la referencia. En la nueva implementación se incluye una llamada al método ClearSecurity() para destruir el vector de permisos que tuviese la referencia previamente, activando así el "salto de protección".

16.6 Modificación de la instrucción Delete

Esta instrucción borra el objeto al que apunta la referencia. Se trata de otra de las instrucciones que se ven afectadas por el mecanismo de protección. En el diseño realizado se ha decidido proteger esta operación siempre que no exista salto de protección, es decir, si se cuenta con permisos para todos los métodos, entonces se podrá también borrar el objeto, si existe algún permiso restringido (referencia restringida), entonces no será posible borrarlo.

Para implementar esto se deberá comprobar la existencia de Salto de Protección (2.1), en cuyo caso se permite el borrado, y en caso contrario se lanza una excepción capturable TSecurity.

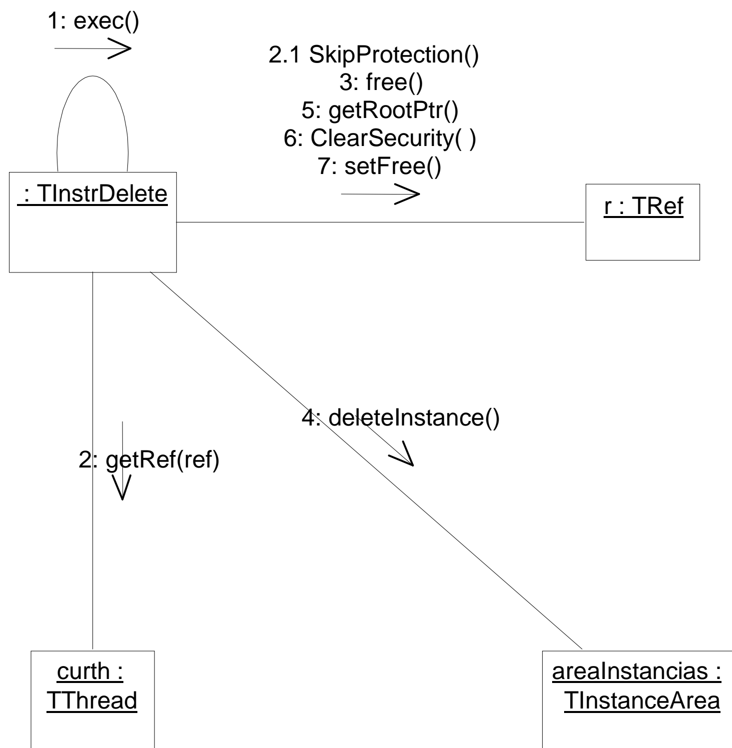


Figura 16.9: Ejecución de la instrucción Delete

Al borrar la instancia se borra también el vector de permisos invocando a ClearSecurity() (6), para así inicializar completamente la referencia y no dejar permisos antiguos en la misma.

17 Análisis del Rendimiento del Sistema con la Incorporación del Mecanismo de Protección

Una vez implantado el mecanismo de protección, se realiza una batería de pruebas con el fin de realizar verificaciones del mecanismo y mediciones de tiempo de sobrecarga que añade dicho mecanismo. En concreto se trata de verificar la sobrecarga que conlleva la comprobación de permisos en la invocación a métodos, que es la operación que más veces se realizará, dentro de las que se ven afectadas por el mecanismo de protección.

Las pruebas realizadas han tenido lugar en un procesador Pentium 200Mhz, ejecutando Linux 2.0. El test intenta comparar el tiempo de ejecución de una invocación a un método en la máquina sin protección frente al tiempo de ejecución de una invocación al mismo método en la máquina que incorpora el mecanismo de protección.

Cuando se introdujo el mecanismo de seguridad en la máquina se tuvo la precaución de diferenciar el nuevo código añadido a la máquina para implementar seguridad del código original de la misma, mediante directivas de compilación condicional. Así, si se desea compilar la máquina con el mecanismo de protección funcionando basta con no incluir la directiva de compilación –DSEC_ENABLED, y se generará código binario para una máquina que funciona sin mecanismo de protección.

Se denominará Máquina Base a la versión original de la máquina, sin mecanismo de protección, en la que la instrucción ForbidExec está presente pero no tiene efecto. La segunda versión de la máquina (Máquina Protegida) es la nueva máquina, evolución de la anterior, que implementa el mecanismo de protección. El diseño e implementación de estos primeros prototipos no ha sido efectuado con la eficiencia como objetivo primario, pudiéndose incluir fácilmente muchas optimizaciones.

El programa de prueba que se ha preparado es un bucle que repetidamente ejecuta la invocación a método que va a ser comprobada. Para cada tipo de método (invocación a método de la clase Integer, Float, método vacío de usuario y método arbitrario de usuario) se efectúan dos ejecuciones. Una ejecución con el “Salto de Protección” activado, lo que hará que no sean comprobados los permisos sobre el método y permitirá evaluar la velocidad máxima de la máquina protegida en la invocación a un método. La otra ejecución comprueba los permisos de acceso, para lo cual se ejecuta la instrucción ForbidExec sobre la referencia al objeto sobre el que se hace la invocación, y permitirá evaluar el costo adicional de la comprobación de permisos. Además, se realiza la misma ejecución con la Máquina Base para determinar la sobrecarga que conlleva la comprobación de permisos. Se realizan varias ejecuciones y se toma la media de las mismas.

17.1 Creación de una Clase primitiva Reloj para evaluar el rendimiento

La máquina abstracta original no contiene en su diseño ninguna conexión con la máquina física exceptuando la Entrada/Salida. Para la realización de pruebas de rendimiento de la implementación se creó una nueva clase primitiva, Clock, que nos permitirá cronometrar porciones de código.

```
Class Clock
Isa Object
Methods
  Reset();
  GetTime():Float;
EndClass
```

La implementación de estos métodos se basó en la función de librería ANSI C, clock(). Esta función devuelve un contador de pulsos de reloj. Haciendo diferencias entre los valores obtenidos por esta función y conociendo la relación entre pulsos y tiempo real puede calcularse el tiempo transcurrido en segundos.

Al instanciar un objeto de tipo Clock se inicializa su cuenta. Al invocarse el método GetTime() se devuelve el número de segundos transcurridos desde el último Reset() o su creación.

17.2 Pruebas de rendimiento

En el siguiente cuadro se muestra el esqueleto general del programa que se utilizó para realizar las pruebas, lo que también sirve para dar una idea de cómo es un programa en Carbayonia.

Las diferentes invocaciones a métodos comprobadas son las siguientes:

- Métodos de un entero
- Métodos de un real
- Método vacío de usuario
- Método arbitrario de usuario

El tiempo que se tarda en la ejecución de un bucle vacío se utiliza como base para medir el tiempo de ejecución de una invocación a método.

```
Class MyApp

//Métodos de la clase
Methods

run() //Punto de arranque
Code
  this.test;
  Exit;
EndCode


test() //Método programa prueba

Refs //Sólo se crean referencias
  b: bool; //para saltos condic.

Instances
//Los objetos de estas referencias
//se crean al entrar en el método
  clk: clock;
  elapsedTime: float;
  zero: integer(0);
  one: integer(1);
  max: integer(50000);//no. bucles
  i: integer; //índice bucle
  //consola para escribir resultados
  c: constream;
  msg1: string('Tiempo: ');

  //Objeto sobre el que se
  //comprobará la llamada a método

  mTest: ClaseUsuario;

Code
//Para obligar a comprobar permisos
//se hace ForbidExec a un método
//distinto del que se comprueba
//Por omisión el salto de protección
//está activo, por lo que la línea
//siguiente se comenta para suprimir
//la comprobación de permisos

  ForbidExec mTest, metodo2; // (1)

Initialization:
  i.set(zero);

  clk.reset(); // Inicio medición

Loop:
  i.less(max):b;

//Salta al final si b es Falso
//(i no es menor que el número
//máximo de bucles

  JF b, LoopEnd;


//Método que se prueba, en este caso
//un método de usuario
//La siguiente línea se reemplaza
//por el método concreto a probar

  mTest.metodoAProbar(); // (2)

  i.add(one);

  Jmp  Loop;

LoopEnd:

//obtener tiempo transcurrido e
//imprimir en consola

  clk.getTime():elapsedTime;

  c.write(msg1);
  c.write(elapsedTime);
  c.nextLine();

  Exit;

EndCode

EndClass
```

Este programa realiza 50.000 iteraciones sobre la instrucción indicada como (2) en el código, devolviendo por pantalla el tiempo total de ejecución del bucle.

La instrucción (1) se incluirá en el código si queremos medir el tiempo de ejecución con comprobación de permisos. Al realizarse el ForbidExec se elimina el salto de protección (que está activo por omisión) y se obliga a la comprobación de permisos en la invocación de métodos. Si se invocase la instrucción (2) precisamente con el método que se restringe en el

ForbidExec se produciría una excepción de seguridad, pues no se tendría permiso para invocar ese método. Colocando un método diferente en (2) sí se tendrá permiso y así se evaluará el tiempo que tarda con comprobación de permisos.

17.2.1 Resultados

Para la realización de los tests se utilizó el siguiente equipo:

- **Equipo Hardware**: Procesador Pentium 200 MHz, 64 Mb RAM, Disco duro 6,4 Gb
- **Sistema Operativo**: Linux Suse 5.3 (Linux 2.0)

Se implementaron cuatro versiones del código anterior con el objetivo de medir los tiempos de ejecución de los siguientes métodos:

- Método de la clase Integer.
- Método de la clase Float.
- Método de una clase de usuario vacío. Sólo contiene la instrucción Exit.
- Método de una clase de usuario con instrucciones arbitrarias. Su complejidad es mayor que los anteriores para dar una idea de la complejidad de un método de usuario en la práctica.
- Bucle vacío. Se midió este tiempo para restar este valor de los anteriores y así obtener el tiempo exacto de ejecución del método evaluado.

Los resultados de estos experimentos se muestran en las siguientes tablas. En la tabla que se muestra a continuación se incluyen los valores absolutos del tiempo total de ejecución del programa de prueba.

		Máquina Base	Máquina Protegida
Con *Salto de Protección*	Bucle vacío	31,6934	31,7393
	Entero	41,4425	41,5060
	Real	43,7715	43,5168
	Método vacío	44,5862	44,6500
	Método de usuario	523,8695	522,4745
		Máquina Base	Máquina Protegida
Comprobando permisos	Bucle vacío	31,7017	31,5026
	Entero	41,5140	42,6888
	Real	43,7827	46,1041
	Método vacío	44,4923	47,3936
	Método de usuario	523,6435	524,7465

Tabla 17.1: Tiempos totales de ejecución para el código evaluado

El primer bloque de la tabla corresponde a la invocación de métodos sobre una referencia a un objeto que tiene Salto de Protección, es decir, que existe permiso para todos los métodos y por tanto el mecanismo de protección no los comprueba. Este mecanismo únicamente comprueba que existe salto de protección. En el caso de la Máquina Base, no está implementada la protección, por lo que ni siquiera se examina si hay salto de protección o no.

Se puede comprobar, en los resultados, que no existe prácticamente diferencia entre ambas máquinas, es decir, que el mecanismo de protección apenas sobrecarga cuando se ejecutan invocaciones a métodos sobre objetos no protegidos (salto de protección activo).

El segundo bloque de la tabla corresponde a la invocación de métodos sobre referencias (capacidades) que tienen restricciones de protección, puesto que en algún momento se ha invocado la instrucción ForbidExec que restringe algún permiso. En este caso, el mecanismo de invocación de la máquina debe comprobar la existencia de permisos. Esto debería suponer una sobrecarga adicional en la máquina.

Las primeras dos columnas de la tabla siguiente muestran el tiempo que tarda la ejecución del método que se prueba. Este tiempo se obtiene restando al tiempo total de ejecución del programa el tiempo que tarda un bucle vacío. Las siguientes columnas muestran el incremento en el tiempo de ejecución de un método en la máquina protegida frente a la máquina base, tanto en tiempo absoluto como en porcentaje.

		Tiempo de ejecución de método		Incremento de máquina protegida frente a base	
		Máquina Base	Máquina Protegida	Absoluto	Porcentaje
Con *Salto de Protección*	Bucle vacío	0	0		
	Entero	9,7491	9,7667	0,0176	0,181%
	Real	12,0781	11,7775	-0,3006	-2,489%
	Método vacío	12,8928	12,9107	0,0179	0,139%
	Método de usuario	492,1761	490,7352	-1,4409	-0,293%
		Tiempo de ejecución de método		Incremento de máquina protegida frente a base	
		Máquina Base	Máquina Protegida	Absoluto	Porcentaje
Comprobando permisos	Bucle vacío	0	0		
	Entero	9,8123	11,1863	1,3740	14,002%
	Real	12,0810	14,6016	2,52055	20,864%
	Método vacío	12,7906	15,8910	3,1004	24,240%
	Método de usuario	491,9419	493,2440	1,3021	0,265%

Tabla 17.2: Tiempos de ejecución para los métodos probados

Es interesante hacer notar que cuando las diferencias en el tiempo de ejecución son muy pequeñas, los porcentajes son aproximados. Esto es debido a que las variaciones aleatorias entre ejecuciones pueden ser mayores que esta diferencia. Otras variaciones pequeñas y posiblemente extrañas entre las máquinas se pueden atribuir a que las máquinas no son exactamente iguales a pesar de usar compilación condicional y a diferencias en las optimizaciones y la generación de código del compilador.

17.2.1.1 Sobrecarga del salto de protección

Como muestran estos datos, la sobrecarga que introduce el salto de protección sobre la máquina base pasa prácticamente inadvertida. De hecho, las variaciones anteriores hacen que en algunos casos se mejore el rendimiento. Cuando de hecho se está ante una sobrecarga, esta es únicamente de un 0,16%.

17.2.1.2 Sobrecarga de la comprobación de permisos

Esta sobrecarga tiene una variación mayor en cuanto a porcentajes. Para llamadas elementales sin complejidad (como las llamadas a enteros, reales o llamadas a métodos nulos), la sobrecarga parece relativamente grande (entre un 14% y un 24%). Esto no es extraño, ya que cuando el cuerpo de un método es muy sencillo, como lo es en estos casos, el tiempo que se necesita para establecer la llamada al método (en el que hay que comprobar los permisos) es alto en porcentaje comparado con el tiempo que se necesita para ejecutar el

cuerpo (que es muy pequeño). Por tanto, el tiempo de establecimiento de una llamada a método se ve incrementado por la comprobación de permisos del orden de esos 14%-24%.

Sin embargo, cuanto el método tiene una complejidad más acorde con la existente en la práctica en aplicaciones “reales” (aunque sólo 50 veces la complejidad de una llamada a un entero), la sobrecarga disminuye a un simple 0,265%. Esto es así puesto que en los métodos normales de usuario el tiempo de establecimiento no es importante con respecto al tiempo que tarda en ejecutarse el cuerpo del método.

La sobrecarga introducida por la protección es en términos absolutos entre 1 y 3. En teoría debería ser siempre la misma, pero dado que se necesitan realizar búsquedas en los vectores de permisos y de métodos, existen variaciones ya que los prototipos actuales no tienen aún un tiempo constante de búsqueda.

17.3 Consideraciones sobre el rendimiento

En conjunto, las cifras son alentadoras. El salto de protección no supone apenas penalización alguna, y la comprobación de permisos tiene una sobrecarga muy baja cuando los métodos tienen una complejidad razonable.

Aun así, existen una serie de aspectos adicionales que hay que considerar, relativos a mejoras de rendimiento absolutas o percibidas que se pueden incorporar.

17.3.1 La protección siempre implica una sobrecarga

De una u otra forma, si el sistema debe presentar algún grado de protección, será necesario que soporte tareas relativas a ello, que inevitablemente introducirán sobrecarga en alguna medida.

Reducir esta sobrecarga tanto como sea posible, es siempre un objetivo, pero alguna sobrecarga siempre va a aparecer.

17.3.2 Optimizaciones en la implementación

Es posible aplicar fácilmente ciertas técnicas de implementación que pudieran eliminar parte de la sobrecarga del salto de protección, mejorando sustancialmente el tiempo de comprobación. Así por ejemplo, el permiso de salto de protección puede ser implementado a bajo nivel como un bit, que se usaría como parte del código de operación de instrucciones. De esta manera, cada instrucción podría tener dos versiones. Una de ellas, con código par, podría realizar primero la comprobación de protección. La otra, con código impar, podría ir directamente al comportamiento de la instrucción, eliminando la necesidad de comprobar el valor del salto de protección en cada invocación a método.

Para estimar el efecto de una optimización sencilla en la comprobación de permisos, se creó una nueva versión de la máquina protegida (Máquina Protegida Optimizada). La optimización que se implementó fue simplemente una mejor implementación de los permisos (TPerms) En lugar de usar una lista de valores booleanos como en el prototipo previo, se utilizó un vector de bits usando valores long. Se hicieron también algunos cambios menores, como cambiar algunas llamadas por llamadas inline. La tabla 16.3 muestra los resultados de estas mejoras.

		Mejora de la Optimizada frente a la Protegida	
Con *Salto de Protección*		Absoluta	Relativa
	Entero	0,2828	1606,818%
	Real	0,1709	56,870%
	Método vacío	0,1537	856,267%
	Método de usuario	1,7219	119,505%
		Mejora de la Optimizada frente a la Protegida	
Comprobando permisos		Absoluta	Relativa
	Entero	0,2787	20,288%
	Real	1,4015	55,601%
	Método vacío	0,6527	21,051%
	Método de usuario	0,8501	65,291%

Tabla 17.3: Diferencias en los tiempos de Ejecución para métodos de prueba en la Máquina Protegida Optimizada

Los cambios como las llamadas inline deberían ser los responsables de la mejora en los tiempos del salto de protección, los cuales son considerables en porcentaje. Para la comprobación de permisos, las mejoras están en un rango entre el 20% y el 65%. El efecto de estas sencillas optimizaciones indica que las cifras pueden reducirse aún más aplicando más optimizaciones, como por ejemplo la utilización de búsquedas *hash* o mediante la reordenación de las llamadas internas de la implementación hechas por la instrucción Call para favorecer la comprobación de protección.

17.3.3 Número reducido de llamadas protegidas necesario en aplicaciones reales

Un análisis de las aplicaciones reales y los requerimientos de protección indicaría que el número de invocaciones a método en las que es necesaria la comprobación de permisos no sería muy alto, comparado con el número total de métodos en los que no se necesitase protección, (los cuales usarían Salto de Protección). Por tanto el costo de la protección (0,265%-24% por cada llamada a método con estas cifras) sólo tendría que pagarse para una pequeña fracción de los métodos.

Hay que tener en cuenta que la mayor parte de las operaciones protegidas serán operaciones sobre objetos de usuario, no básicos y no locales. Los objetos básicos habitualmente actúan como objetos locales[1] a los métodos, los cuales no necesitan protección, o lo que es lo mismo, tendrán el Salto de Protección activo. Esto evita la comprobación de protección en infinidad de objetos de granularidad fina, y por lo tanto, el impacto del mecanismo de protección se reduce aún más.

Teniendo en cuenta que en el sistema se realizan continuamente invocaciones a métodos, el número de operaciones que deben ir protegidas es relativamente pequeño comparado con el número de operaciones totales. Sólo cuando se desee invocar objetos que no son locales se necesitará protección y esto en general sucederá en contadas ocasiones.

Las evaluaciones anteriores sirven para medir la sobrecarga de protección producida en una invocación a un método. Aunque en la tabla 17.1 se miden sobrecargas absolutas frente al código, esta medida no resulta realista, puesto que en un código normal no se producen siempre invocaciones con protección, habrá una buena parte de ellas que serán sin protección, y las que sean con protección, la mayoría serán para invocar métodos complejos del usuario.

[1] Locales no en el sentido de que residan en la misma máquina, si no en el sentido de que no son compartidos por objetos pertenecientes a otra aplicación.

Todo ello implica que el impacto global de la protección se diluye considerablemente en un programa normal de usuario.

17.3.4 Complejidad de los métodos protegidos en aplicaciones reales

Siguiendo con una argumentación en la misma línea, la complejidad de los métodos que necesitan protección en aplicaciones reales es mucho mayor que la de los métodos simples, como la adición en un entero. Por ello, la sobrecarga de protección tendrá un impacto pequeño para esos métodos, puesto que el tiempo de ejecución será mucho mayor que el tiempo de establecimiento de la invocación. Es decir, la sobrecarga probablemente estará más bien en el rango del 0,265%, correspondiente a una llamada compleja de usuario, que en el rango del 14%, de una invocación a un entero.

17.3.5 Técnicas de análisis estático

Es posible aplicar técnicas de análisis estático al código para reducir aún más la sobrecarga introducida por la comprobación dinámica de permisos. Cuando no cambia la capacidad y la invocación a un método en una sección de código, pueden comprobarse los permisos por adelantado una única vez. Si éstos son correctos, entonces se activa el salto de protección, de forma que en pasadas sucesivas por ese trozo de código ya no se necesitará comprobar los permisos en la invocación al método.

17.3.6 Comparaciones de rendimiento con la arquitectura de seguridad de Java

Estas reflexiones sobre la protección, su rendimiento y cómo mejorarlo podrían aplicarse a otros sistemas similares, como la máquina virtual de Java. En [GS98] se publican algunas cifras sobre el rendimiento de la arquitectura de seguridad de JDK 1.2, comparando el coste de una llamada normal con el coste de una llamada protegida (tal y como la define esta arquitectura) para varios métodos. Aunque la máquina de Java y su mecanismo de protección son diferentes a los descritos en esta tesis, pueden extraerse algunas conclusiones ya que las "llamadas protegidas" del Java pueden considerase análogas a las llamadas "comprobando permisos" descritas anteriomente.

Dependiendo de la complejidad de la llamada, la sobrecarga va desde el 244% para una llamada a método nulo, descendiendo hasta el 81%, 37%, 10%, y finalmente a menos del 0,3%. Este 0.3% es para un método que tiene sobre 1000 veces la complejidad de una llamada a un método nulo.

La sobrecarga para un método complejo es del mismo orden que la medida en la máquina abstracta Carbayonia. Por tanto, parece que las consideraciones y datos anteriores pueden ser tomados como una referencia estimada de la sobrecarga introducida en un sistema por un mecanismo de protección basado en capacidades.

17.3.7 Coste real y aceptación por los usuarios

Teniendo en cuenta estas consideraciones, parece que el coste que introduce la protección propuesta, basada en capacidades, no es alto, teniendo en cuenta la simplicidad, uniformidad y las ventajas de esta aproximación. El coste se reduce enormemente debido a las mencionadas optimizaciones que se pueden introducir, al número reducido de llamadas a objetos que necesitan protección en aplicaciones reales (y los objetos locales, normalmente básicos no tienen penalización) y además a que estas llamadas serán normalmente a métodos con una complejidad para la cual la sobrecarga de la comprobación de permisos es irrelevante.

Teniendo en consideración, además, que la comprobación de permisos sólo aumenta el tiempo de establecimiento de una llamada a método, y no el resto del procesamiento, y que este tiempo de procesamiento en aplicaciones reales suele ser inmensamente superior al de establecimiento, el costo real que se paga por esta protección puede considerarse irrisorio.

Aunque se paga un pequeño precio por el uso de capacidades, es lógico suponer que los usuarios de nuevo sacrificarán un poco de velocidad por la conveniencia que se ofrece, al igual que cuando se utilizan máquinas virtuales en lugar de código compilado[1].

[1] Aunque en este caso la sobrecarga introducida es incluso mucho mayor que la que introduce este modelo de protección.

18 Elaboración de un Entorno Operativo sobre la Máquina Abstracta

18.1 Introducción

Una de las ventajas del mecanismo básico de protección ideado, y que ya ha sido comentada en el apartado correspondiente, es la flexibilidad que este mecanismo aporta al sistema de seguridad. A partir del mecanismo básico puede ser implementada cualquier política de seguridad, adaptándose así a las necesidades particulares del momento o del lugar en el que se pretenda utilizar el mecanismo.

En el capítulo 13 se profundiza en este aspecto, y se hace una descripción de algunas políticas de seguridad que podrían aplicarse, confirmando con ello la flexibilidad del núcleo de protección. En este capítulo se describirá un ejemplo de implementación concreta que muestra cómo utilizar el mecanismo de protección por parte de un servicio de seguridad implantado al nivel de sistema operativo. Este servicio de seguridad proporcionará un segundo nivel de protección, que permite manipular las capacidades a nivel de interfaz de usuario.

Debe recalcarse que este entorno se introduce únicamente como muestra de las posibilidades del mecanismo de protección y del sistema en general. Por tanto muchos aspectos no están pulidos ni finalizados y hay muchos elementos que se pueden mejorar.

18.2 Servicio de seguridad

El modelo de seguridad que se pretende implantar con este ejemplo es muy simple. Se trata de permitir que los usuarios puedan pedir capacidades para objetos (o clases) de los que sólo conocen su nombre simbólico. A partir del nombre, el sistema les devuelve su capacidad. Pero además, los usuarios tendrán la posibilidad de dejar objetos (nombre-capacidad) en un lugar público para que otros usuarios puedan usarlos. Estos objetos podrán tener sus permisos restringidos para que sólo sean accesibles ciertas operaciones.

El servicio de seguridad es en realidad un servicio de denominación. Desde la perspectiva clásica de los sistemas operativos, se asemeja a un sistema de ficheros. En este caso no se guardan ficheros sin cualquier tipo de objeto, pero puede utilizarse una estructura de directorios, semejante a la usada en los sistemas tradicionales.

Para implantar este servicio, se ha desarrollado un sistema algo más amplio, que permite hacer uso de él. Todas las funciones del sistema formarán parte del sistema operativo. Sin embargo, dado que no se han implementado todas las funciones que pudiera tener un sistema operativo, se ha decidido denominar al sistema Entorno Operativo.

18.3 Entorno Operativo

La idea consiste en desarrollar un entorno en el que existan usuarios que puedan conectarse a la máquina. Asimismo existirá un sistema de directorios, donde los usuarios pueden guardar nombres simbólicos de los objetos, junto con las referencias a ellos. Cada usuario tendrá un directorio privado y existirá también un directorio público de acceso común para todos los usuarios.

El acceso al sistema proporcionará al usuario un *shell* sobre el que podrá realizar operaciones relacionadas con accesos a los directorios, cambios de permisos en las capacidades y creación e invocación de operaciones de los objetos.

El usuario podrá crear objetos dentro de su directorio personal, restringir sus capacidades y trasladarlos al directorio público para que otros usuarios puedan usarlos. La posibilidad de invocación de operaciones sobre objetos que tienen capacidades restringidas permitirá comprobar que el mecanismo de protección diseñado funciona correctamente, tanto en el caso de que la operación sea permitida como en el que sea denegada.

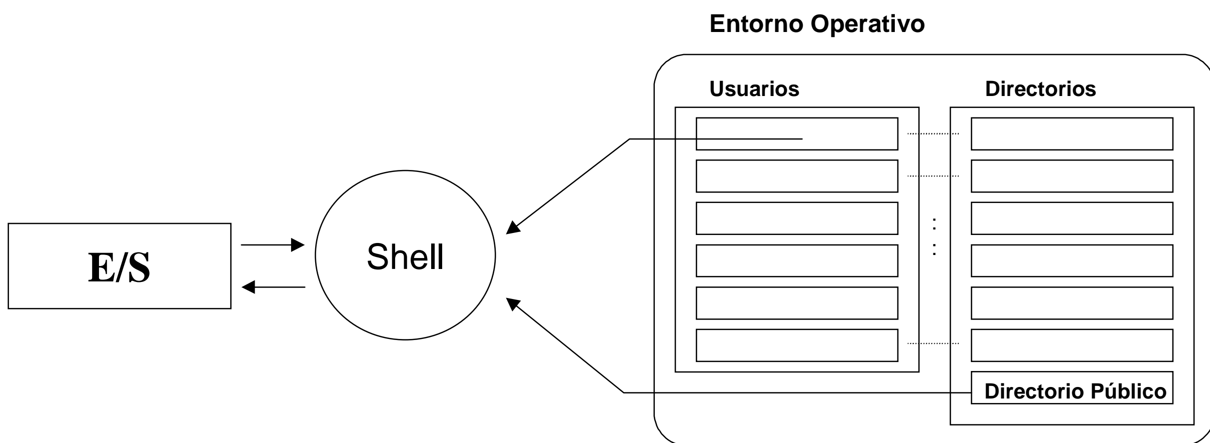


Figura 18.1: Estructura propuesta para el Entorno Operativo

18.4 Requisitos de diseño del Entorno Operativo

La labor de diseño del Entorno Operativo ha sido bastante compleja por la necesidad previa que hubo de ampliar y modificar aspectos de la implementación de la máquina abstracta Carbayonia para poder abordar el diseño del entorno en condiciones óptimas.

La siguiente es una breve descripción de los requisitos impuestos al entorno:

- **Seguridad**. El entorno deberá exhibir el funcionamiento del mecanismo de seguridad diseñado. Deberá usar el mecanismo de seguridad internamente y dar al usuario la capacidad de actuar sobre los permisos de los objetos desde la consola.
- **Persistencia**. Sería interesante disponer de algún mecanismo de persistencia. El esquema de persistencia perseguido permitiría leer y escribir el estado de la máquina en un archivo físico, de modo que se pudiese congelar la ejecución de la máquina y más tarde volver al punto en el que estaba. De esta manera se conservan los objetos creados en los ejemplos entre sucesivas ejecuciones, lo cual es más cómodo.
- **Recolección de Basura**. La velocidad de ejecución de programas grandes en el intérprete disminuye a medida que las áreas crecen de tamaño (al estar implementadas mediante vectores con búsqueda lineal en estos momentos). Esto da lugar a una ralentización apreciable del sistema al aumentar el tamaño de estas áreas. Para mejorar

el rendimiento se pretende diseñar un sencillo mecanismo interno de la máquina que elimine las instancias libres.

- **Ejecución dinámica de instrucciones**. La actual arquitectura de la máquina no dispone de mecanismos para la ejecución dinámica de instrucciones. Sería deseable que el usuario pudiese ejecutar métodos de las instancias que él mismo crea y manipular los permisos de ejecución de dichas instancias.
- **Enlace con la parte física de la máquina**. Otra carencia de la máquina original es que no dispone de un mecanismo estándar de comunicación con las áreas internas de la máquina ni con el exterior (dispositivos físicos). Se estudia la posibilidad de incluir una referencia global reflectiva *System* que permita al sistema operativo relacionarse con los objetos físicos de la máquina.

De todos los objetivos expuestos, en este documento sólo se comentarán dos: la definición del enlace con la máquina abstracta y el diseño e implementación del Entorno Operativo propiamente dicho.

18.5 Definición de un enlace del usuario con la máquina abstracta. Clase System

Esta clase fue creada para permitir a los usuarios del sistema obtener información del funcionamiento interno de la máquina e invocar métodos especiales que modifican su funcionamiento. Todo ello es una forma inicial de reflectividad en el sistema, que permite extender la funcionalidad de la máquina abstracta, como se describió en el capítulo 5.

En principio fue creada para eliminar la instrucción Halt del juego de instrucciones y cambiarla por una llamada a método (system.Halt). Más adelante fue utilizada para incluir otros métodos que accedían a las áreas de la máquina y que se necesitaron para poder desarrollar fácilmente el entorno operativo.

Para dotar a un sistema operativo de un puente real con el mundo físico se debería de completar con métodos que accediesen a dispositivos de almacenamiento, dispositivos gráficos, etc. En definitiva, todos los recursos físicos de la máquina abstraídos mediante una clase instanciada globalmente por el intérprete.

La implementación actual de esta clase tiene los siguientes métodos:

- **Halt**. Sustituye a la instrucción Halt definida originalmente. Almacena el estado de la máquina en el archivo pers.000. y devuelve el control al sistema operativo.
- **GarbageCollect**. Invoca al recolector de basura, liberando instancias no referenciadas.
- **GetNumInst**. Obtiene el número de instancias del área de instancias. Se introdujo pensando en un posible proceso del sistema operativo que necesitase saber el número de instancias creadas en la máquina para así decidir cuando invocar a la recolección de basura.
- **LoadClass**. Carga una clase en el área de clases. Se debe pasar el nombre del archivo con extensión obj. (por ejemplo, para cargar la clase demo.asm, debe invocarse al método como LoadClass("demo.obj").
- **UnLoadClass**. Descarga una clase del área de clases. El nombre se pasa sin extensión y en mayúsculas.
- **GetNumClasses**. Obtiene el número de clases cargadas en el área.

- **GetClassName(k).** Obtiene el nombre de la clase k. Es usado en conjunto con el método anterior para obtener los nombres de todas las clases del sistema.

Esta clase no será instanciada por los usuarios sino que será la máquina quien cree una instancia de esta clase y la inicialice adecuadamente. Se define una referencia global nueva de la máquina denominada SYS. Esta referencia se crea al crear cada hilo de la máquina y apunta siempre a la instancia creada por el sistema operativo. En esta primera implementación no se ha introducido ninguna restricción en la ejecución de los métodos de la misma, pero sería sencillo pasar una referencia restringida SYS a los usuarios no privilegiados del sistema. Este es un ejemplo más de la uniformidad que presenta la protección basada en capacidades diseñada para el sistema: todos los elementos se protegen por igual, incluso instancias básicas definidas por la propia máquina.

18.6 Otras modificaciones

Además, se realizaron otros cambios necesarios para posibilitar la implementación de un entorno operativo. Fundamentalmente se necesita ejecutar dinámicamente instrucciones que teclea el usuario en la consola. Como ejemplo de estas modificaciones está la introducción de métodos adicionales en la clase Object para saber la clase a la que pertenece el objeto, el número y nombre de los métodos de la clase, etc. Esto permite al usuario consultar dinámicamente las clases que existen en el sistema (con la clase System), los métodos que tiene la clase, etc. Como se ve, es una forma de reflectividad estructural necesaria para lograr el objetivo anterior.

Así mismo, se introdujeron instrucciones adicionales. Una de ellas ya estaba prevista (aunque aún sin implementar), para permitir crear una instancia de una clase que se indica mediante una cadena. Otra permite componer en una cadena el texto de una instrucción y ejecutarla dinámicamente. Así el usuario puede, sobre la marcha, construir expresiones para crear un objeto de la clase que desee, invocar uno de sus métodos, etc. El siguiente es un pequeño ejemplo de esto.

```
CreaReferencia (cl:string):object // Devuelve referencia clase 'cl'
Refs
 robj:object;  // es genérico
Instances
 str:string;
Code
 str.set('new robj, '); // Inicializa parte fija de la instrucción
 str.insert(str, cl); // Concatena el parámetro
 Eval str;            // Interpreta la instrucción 'new robj, cl' y
                      // crea una instancia de la clase 'cl'.
 Assign rr, robj;  // devuelve una referencia a la clase.
 Exit;
EndCode
```

Una manera más ambiciosa de lograr este mismo objetivo sería desarrollar e implementar de manera sistemática y en profundidad una arquitectura reflectiva para la máquina en sus dos vertientes: estructural y de comportamiento. Para la parte estructural, se trataría de exponer completamente la estructura de los programas Carbayonia (clases, instancias, referencias, métodos, instrucciones de los métodos, etc.). En esta aproximación más modesta, esto se logra parcialmente y exclusivamente en los casos de interés para el problema incluyendo métodos en la clase Object.

En la otra vertiente se trataría de exponer el funcionamiento interno de la máquina (área de clases, área de instancias, área de referencias, hilos, etc.). Obviamente, además de poder consultar esta información, la reflectividad permitiría modificarla. En esta aproximación se introduce la clase reflectiva System con algunos métodos relacionados con este aspecto.

Sin embargo, la construcción de una completa arquitectura reflectiva para la máquina se sale del ámbito de trabajo de esta tesis, por lo que se ha optado por esta otra solución más específica y adaptada a las necesidades concretas que se detectaron. En [Fer99] puede encontrarse un desarrollo de este estilo para una versión de la máquina que no incorpora características de protección, cuya implementación se solapa en el tiempo con la presentada aquí.

18.7 Características generales del Entorno Operativo[1]

El Entorno Operativo cuenta con las siguientes características:

- Los usuarios del sistema no tienen contraseña de entrada, por simplicidad y se conectan al sistema inicialmente (proceso de *login*)
- El único usuario creado inicialmente en el entorno es el superusuario (*root*). Al igual que en Unix, este usuario tiene más capacidades que los usuarios normales. En este caso, se le han asignado las funciones de mantenimiento de usuarios y de invocación de un sencillo mecanismo de persistencia que guarda el estado del entorno en disco. Sin embargo, y al contrario que Unix, no es omnipotente en el sistema. Es simplemente una abstracción, un objeto que tiene autoridad para ese mantenimiento de usuarios, pero no para otras funciones del sistema. Por ejemplo, no puede acceder a voluntad a los objetos de otros usuarios.
- Cada usuario tiene asociado un directorio personal que se crea y se destruye con el usuario. Los directorios que se van a manejar en este entorno son similares a los directorios convencionales de sistemas de ficheros pero con la particularidad de que almacenan instancias de clases de la máquina y no ficheros. Los propios directorios son instancias de una clase Directorio.
- Cada directorio está formado por un conjunto de elementos denominados *Slots* (entradas de directorio). Cada una de estas entradas permite asociar un nombre simbólico a una referencia (que apuntará a un objeto determinado). En esencia, permite asignar un nombre simbólico a un objeto. Por otro lado, no se implementan órdenes del *shell* para la manipulación de subdirectorios aunque sería una ampliación fácil de desarrollar. Sin embargo, la estructura de directorios no es plana, puesto que en un directorio se puede almacenar cualquier objeto, incluyendo instancias de la propia clase Directorio.
- Existe un directorio público común a todos los usuarios donde pueden realizar las mismas operaciones que sobre su directorio privado. Este es el medio de comunicación entre usuarios y que permitirá exhibir el funcionamiento de los mecanismos de protección. Puede pensarse en este directorio como un tablón de anuncios compartido por todos los usuarios en los que cada usuario expone los objetos (añade referencias asociadas a nombres simbólicos) que desea que puedan ser accedidos por cualquier otro usuario. Evidentemente, el usuario propietario del objeto

[1] En [Gar99] se encuentra el documento original de la implantación del entorno. Este documento se centrará en los aspectos más relevantes.

restringirá previamente los permisos del objeto que está en una entrada determinada a las operaciones que el público en general puede hacer con el objeto, antes de publicar esa entrada en el directorio público.

- Existe un *shell* para introducir órdenes en la línea de comandos para facilitar las tareas al usuario, similar al *shell* del sistema operativo Unix.

18.8 Órdenes del Shell

A continuación se describen las órdenes del *shell* que podrá utilizar el usuario para interactuar fácilmente con el sistema y probar el funcionamiento del mecanismo de protección (la otra es mediante programación normal). Estas se han agrupado en tres categorías:

- **Órdenes internas**. Permiten realizar tareas generales dentro del sistema: Cargar y descargar clases, listar las clases existentes, salir del shell, llamar al recolector de basura y cerrar la sesión de trabajo con el sistema.
- **Gestión de usuarios**. Permiten añadir y eliminar usuarios, listar todos los usuarios definidos y saber cuál es el usuario activo.
- **Gestión de *slots* y directorios**. Son las que permiten realizar la demostración de las propiedades de protección del entorno. Por una parte permiten ver la información que hay en los directorios: listar las entradas en el directorio, obtener información detallada de cada entrada, etc. Por otro lado permiten crear nuevos objetos que se añaden a un directorio (dándoles un nombre simbólico), eliminar entradas, cambiar el nombre de una entrada, publicar una entrada en el directorio público.

  El último subgrupo de órdenes permite trabajar con los objetos de un directorio: invocar uno de sus métodos o bien restringir la ejecución de uno de sus métodos. Esta última orden hará que cualquier intento subsecuente de invocación del método produzca una excepción de protección.

18.8.1 Órdenes internas

LDCLASS <clase.obj>

Descripción: Carga la clase especificada a través de su fichero de clases en el área de clases.

Notas: La clase debe haber sido compilada previamente con el compilador de clases. Debe especificarse siempre el nombre del archivo con extensión obj.

Ejemplo: `$ LDCLASS bench.obj`

ULCLASS <clase>

Descripción: Descarga la clase especificada del área de clases.

Notas: El nombre de la clase debe teclearse en mayúsculas y sin extensión.

Ejemplo: `$ ULCLASS BENCH`

LSCLASS

Descripción: Lista por pantallas las clases cargadas en el área de clases.

Ejemplo: `$ LSCLASS`

HALT

Descripción: Realiza una copia persistente de las áreas de la máquina en disco y devuelve el control al sistema operativo.

Notas: El estado de la máquina se almacena siempre en un archivo denominado pers.000. Para restaurar el estado de la máquina simplemente hay que invocar al intérprete con el signo menos como único parámetro: `$ io3 -`

GRBCOL

Descripción: Invoca al método de recolección de basura[1] que elimina del área de instancias todas las instancias libres. Es recomendable utilizarlo cuando se observe una ralentización en el funcionamiento del entorno.

EXIT

Descripción: Finaliza la ejecución del *shell* y vuelve a la pantalla de petición de conexión al sistema (*login*).

18.8.2 Gestión de Usuarios

ADDUSR <nombre>

Descripción: Añade un usuario a la lista de usuarios.

Notas: Esta orden sólo puede ser utilizada por el usuario '*root*'.

Ejemplo: `$ ADDUSR guest`

DELUSR <nombre>

Descripción: Borra un usuario de la lista de usuarios.

Notas: Esta orden sólo puede ser utilizada por el usuario '*root*'

Ejemplo: `$ DELUSR guest`

LISTUSR

Descripción: Lista por pantalla los usuarios del sistema.

WHOAMI

Descripción: Muestra por pantalla el usuario activo.

[1] Una recolección de basura elemental incluida únicamente con el fin de facilitar el uso de este entorno de ejemplo.

18.8.3 Gestión de slots y directorios

Las órdenes que se listan a continuación permiten la manipulación de *slots* y directorios. Como existe un directorio público a todos los usuarios, se implementa una versión alternativa de todas las instrucciones que permite realizar la operación sobre el directorio público. Esto se realiza mediante el indicador @p como primer parámetro de la instrucción.

DIR / DIR@P

Descripción: Lista por pantalla los *slots* del directorio. Si se utiliza la segunda versión de la orden lista los *slots* del directorio público.

NEW [@]p<nombre> <clase>

Descripción: Crea un nuevo *slot* en el directorio con ese nombre e instancia un objeto de la clase solicitada al que se asociará el nombre a través de una referencia.

Notas: La clase indicada debe estar cargada en el área de clases (véase LDCLASS), y debe escribirse en letras mayúsculas. Si el nombre ya existe se producirá un error.

Ejemplo: `$ NEW slotbool, BOOL`

REN [@p] <nombre_antiguo> <nombre_nuevo>

Descripción: Cambia el nombre de un *slot*.

Notas: El *slot* que se modifica debe existir en el directorio. El nombre nuevo no puede existir previamente.

Ejemplo: `$ REN slotbool, myslot`

DEL [@p] <nombre>

Descripción: Borra del directorio el *slot* especificado.

Ejemplo:

```
$ NEW x, BOOL
$ DEL x
```

MKPUB <nombre>

Descripción: Crea una copia del *slot* especificado en el directorio público.

Ejemplo:

```
$ MKPUB mybool
$ DIR@P
```

VIEW [@p] <nombre>

Descripción: Muestra una descripción detallada del *slot* pasado como parámetro. La información que se muestra es la siguiente:

- Nombre del *slot*
- Clase a la que pertenece
- Métodos de que dispone, valor de retorno de cada método y si es posible ejecutar o no cada método.

Ejemplo:

```
$ view mybool
Slot: mybool
Clase: BOOL
Metodos disponibles:
        0       GETCLASS():STRING SI
        1       GETID():INTEGER SI
        2       IS():BOOL SI
        3       GETNMETH():INTEGER SI
        4       GETMTNAME():STRING SI
        5       GETMTNDX():INTEGER SI
        6       GETMTRV():STRING SI
        7       CANEXEC():BOOL SI
        8       FORBIDEXECUTION():VOID SI
        9       GETMTNPAR():INTEGER SI
        10      GETMTPARTYPE():STRING SI
        11      SETTRUE():VOID SI
        12      SETFALSE():VOID SI
        13      NOT():VOID SI
        14      AND():VOID SI
        15      OR():VOID SI
        16      XOR():VOID SI
```

FEXEC [@p] <nombre> <método>

Descripción: Prohibe la ejecución del método pasado como parámetro a través del *slot* indicado.

Ejemplo:

```
$ FEXEC mybool, XOR
```

EXEC [@p] <nombre> <método>

Descripción: Ejecuta el método especificado a través del *slot*.

Notas: Si el método tiene prohibida la ejecución se produce una excepción de protección. En esta versión del entorno sólo se permite la invocación de métodos sin parámetros.

Ejemplo:

```
$ EXEC mybool, GETID
```

18.9 Diseño e Implementación del Entorno Operativo

En la siguiente figura se muestra el diseño de clases para el entorno operativo, en el que aparecen los elementos esenciales que intervienen en él: el sistema o entorno propiamente dicho, los usuarios, los directorios (que contienen un conjunto de *slots* que apuntan a objetos) y el *shell* que permite el trabajo con el sistema mediante un pequeño conjunto de órdenes.

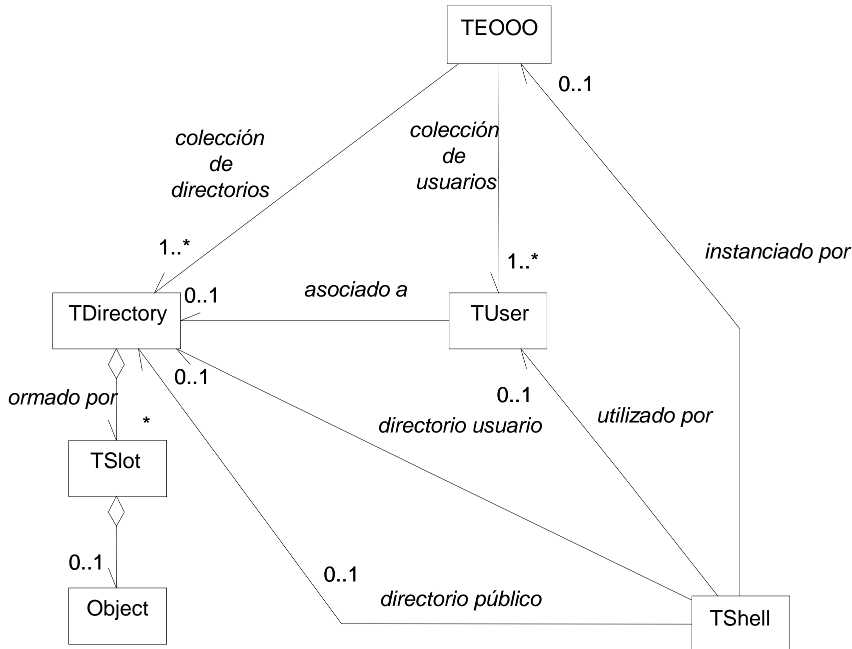


Figura 18.2: Diagrama de clases del Entorno Operativo

Cada clase realizará las siguientes tareas:

- **Clase TEOOO**. Entorno Operativo Orientado a Objetos. Formado por un conjunto de usuarios, un conjunto de directorios y un directorio público. Dispondrá de métodos para manipular las estructuras internas del sistema y dar paso a los usuarios al sistema a través de un método login. Los métodos de manipulación de usuarios estarán restringidos para aquellos usuarios no privilegiados.
- **Clase TDirectory**. Directorio. Mantiene una colección de objetos creados dinámicamente por el usuario (conjunto de entradas o *slots*). Se utiliza para representar el directorio de cada usuario y el directorio público de intercambio.
- **Clase TUser**. Usuario. Representa un usuario del sistema. Al crearse se asocia a una instancia directorio que será su directorio privado.
- **Clase TShell**. *Shell* del sistema. Representa el medio de comunicación de los usuarios con el sistema operativo. Al crearse se construye con un usuario inicial, una referencia al Entorno Operativo que lo instanció y una referencia al directorio público de ese usuario. El Entorno Operativo instancia un *shell* con los parámetros correspondientes al ser llamado por un usuario válido del sistema.
- **Clase TSlot**. Objeto asociado a una referencia. Representa un elemento de un directorio. Sólo contiene un nombre y una referencia genérica a Object (así puede almacenar objetos de cualquier tipo) y los métodos necesarios para manipular estos atributos.

18.10 Diseño detallado

Un diagrama de clases más detallado del entorno, incluyendo métodos y atributos puede verse a continuación.

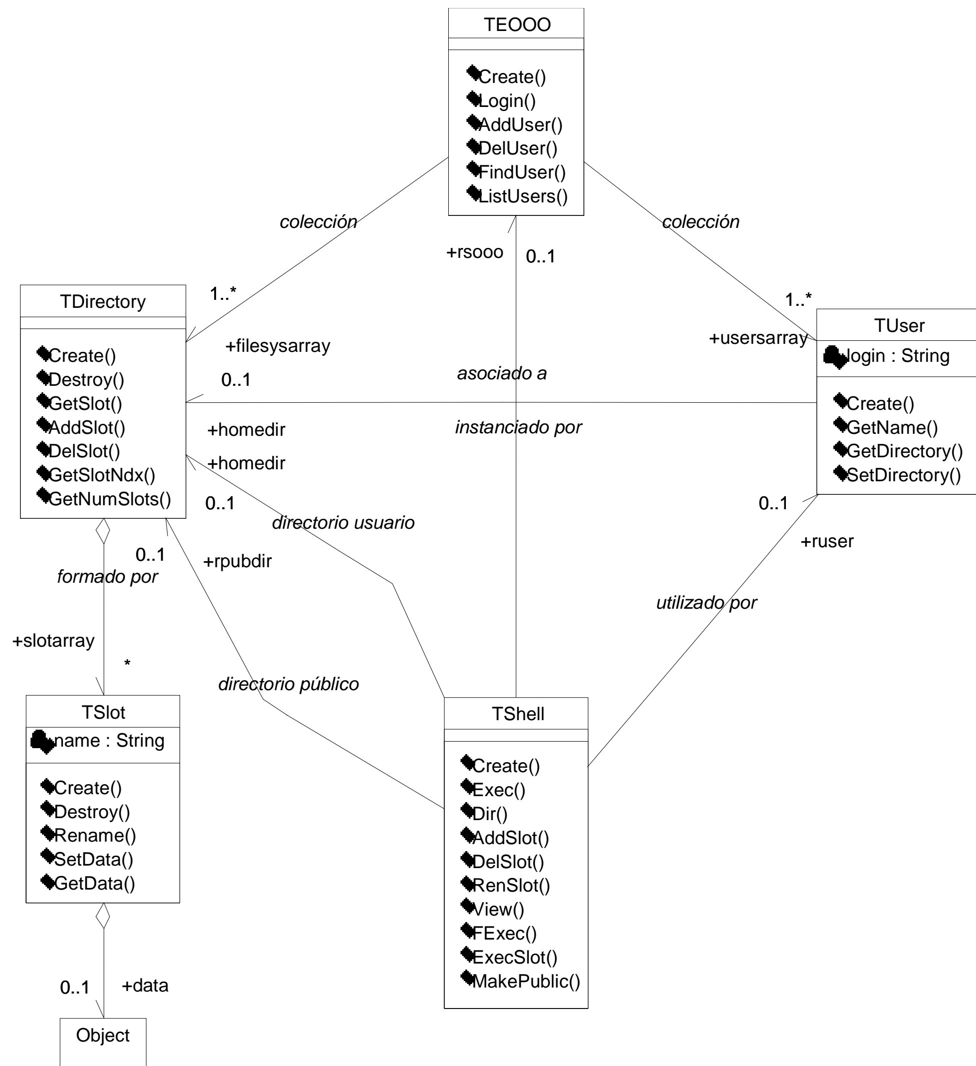


Figura 18.3: Diagrama detallado de clases del Entorno Operativo

En la siguiente figura se muestra el funcionamiento general del Entorno Operativo mediante un diagrama de colaboración.

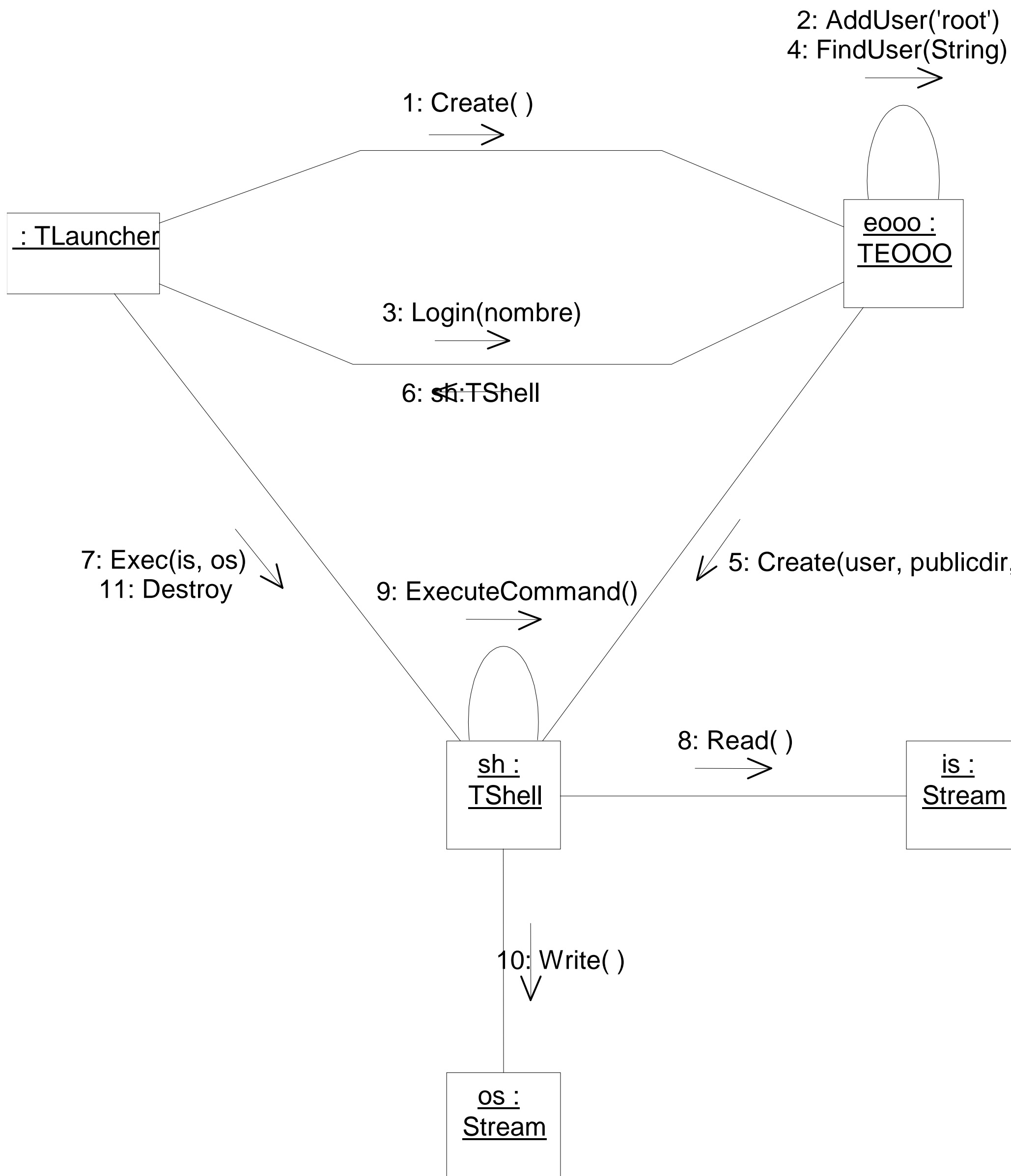


Figura18.4: Funcionamiento general del Entorno Operativo.

La instancia de la clase Launcher (lanzador) representa la función principal que instancia el entorno operativo. En primer lugar invoca el método Create() que crea el usuario *Root* (1, 2). A continuación, lee de la consola el *login* del usuario. Con este valor invoca al método login del entorno operativo (3). El entorno busca el usuario en su lista de usuarios (4) y si lo encuentra construye un *shell* y lo inicializa con el usuario, el directorio público del sistema y una referencia al propio entorno operativo que lo crea (5). De esta manera el shell conoce quién es el usuario al que pertenece, el directorio público al que puede acceder y quién es el entorno que lo creó. Esta información la puede utilizar al ir trabajando con el sistema. Cuando el usuario de entrada no es *root*, la referencia al entorno operativo viene con los métodos de gestión de usuarios restringidos (mediante la instrucción correspondiente de la máquina forbidExec). De esta manera el *shell* de un usuario normal no podrá invocar a estos métodos, garantizando que sólo el superusuario pueda gestionar la lista de usuarios del sistema. Este es un ejemplo de utilización de la protección basada en capacidades en la propia programación del entorno de ejemplo.

Cuando el lanzador recibe una referencia al *shell*, invoca el método Exec del mismo, pasándole como parámetros dos flujos *(streams)*: uno de entrada y otro de salida. Esta implementación permite un uso muy genérico del *shell*, que permitiría recibir un archivo de entrada y un archivo de salida y hacer una ejecución por lotes.

El *shell* lee órdenes del flujo de entrada, realiza la acción solicitada y devuelve sus resultados por el flujo de salida. Cuando se introduce la instrucción EXIT, el *shell* termina y se devuelve el control al lanzador, que repite el proceso de entrada al entorno.

18.11 Documentación del Entorno

La documentación de las clases y sus respectivos métodos fue realizada con la herramienta CASE Rational Rose 98. Se ha respetado el formato original de salida para resaltar las bondades del uso de una herramienta CASE en el proceso de documentación.

TEOOO

Entorno Operativo Orientado a Objetos.

El hilo principal de la aplicación instancia el sistema operativo y lo inicializa. Luego invoca a Login pidiendo un nombre de usuario, si existe se crea un *shell* para él.

Operaciones Públicas:

Create () :

Inicializa el array de usuarios y de sistemas de ficheros.

Crea el usuario *'root'* y el directorio publico.

Login (user : string) : TShell

Este método recibe un nombre de usuario como parámetro.

Busca el usuario en la lista de usuarios del sistema y, si es encontrado, crea un *shell* con la referencia al usuario, una referencia al directorio publico y una referencia al propio sistema operativo, que tendrá permisos restringidos para los métodos de manipulación de usuarios si el usuario no es *'root'*.

AddUser (login : string) : bool

Añade un usuario al sistema.

Se comprueba que el usuario no existe, si existe se devuelve un error.

En otro caso, se crea el usuario y un directorio que se asocia al usuario.

DelUser (name : String) : bool

Borra un usuario del sistema.

Localiza el usuario en la lista de usuarios, y si no se encuentra termina.

Borra el sistema de ficheros asociado al usuario y borra el usuario en sí.

FindUser (user : String) : TUser

Busca un usuario en la lista de usuarios, si se encuentra devuelve la referencia asociada al mismo.

ListUsers (os : Stream) :

Lista los usuarios del sistema a través del flujo pasado como parámetro.

TUser

Clase Usuario.

Representa un usuario del sistema.

Atributos Privados:

login : String

Operaciones Públicas:

Create () :

Constructor de usuario.

Crea un usuario con un nombre.

GetName () : String

Devuelve el nombre del usuario.

GetDirectory () : TDirectory

Devuelve el directorio asociado al usuario

SetDirectory (dir : TDirectory) :

Fija el directorio del usuario.

Sólo es invocado tras su creación en AddUser para unirlo a su directorio.

TDirectory

Clase Directorio.

Contiene una colección de *slots* y dispone de métodos para manipularlos.

Operaciones Públicas:

Create ():

Constructor de Directorio.

Inicializa el array de *slots*.

Destroy () :

Destructor de Directorio.

Destruye la colección de *slots*.

GetSlot (name : String, wantdata : Bool) :

Devuelve un *slot* a partir del nombre simbólico.

Obtiene la referencia que contiene el *slot* (si wantdata es Verdadero) o bien el TSlot en sí.

El uso de wantdata se introdujo al codificar el método FExec del *shell*, para restringir la ejecución de un método, que requiere acceder a la referencia física del *slot* y actualizar sus permisos.

AddSlot () :

Añade un *slot* al directorio.

Se pasan como parámetros el nombre simbólico del *slot* y una referencia object con el objeto que refiere.

DelSlot () :

Borra un *slot* del directorio.

Borra el *slot* solicitado del directorio y su referencia asociada.

GetSlotNdx (ndx : Integer) : object

Obtiene la referencia a la que apunta el *slot* que ocupa la posición ndx.

Es utilizado por el método Dir de TShell para iterar sobre la lista de *slots* del directorio.

GetNumSlots () : Integer

Devuelve el número de *slots* del directorio.

Utilizado conjuntamente con GetSlotNdx para iterar sobre los elementos del directorio.

TSlot

Clase *Slot*.

Contenedor de referencias. Asocia un nombre simbólico a una referencia a object.

Atributos Privados:

name : String

Operaciones Públicas:

Create (name : String, ref : Object) :

Constructor de *slot*.

Recibe un nombre simbólico y la referencia que se desea asociar al nombre.

Destroy () :

Destructor de *slot*.

Libera los datos asociados a la referencia si estos no son nulos.

Rename (name : String) :

Cambia el nombre del *slot.*

SetData (datos : Object) :

Fija la referencia del slot a los datos pasados como parámetros.

GetData () : Object

Devuelve la referencia a los datos del slot.

TShell

Shell del entorno operativo.

Instanciado por el sistema operativo al recibir una operación de *login* correcta.

El que lo recibe ejecuta el método Exec para interactuar con el *shell*, pasándole como parámetros los flujos de E/S que desee.

Operaciones Públicas:

Create (sooo : TSOOO, publicdir : TDirectory, user : TUser) :

Inicializa el objeto *shell* a partir de los parámetros.

Obtiene el directorio de trabajo del usuario, inicializa el directorio publico e inicializa la referencia al sistema operativo.

Exec (is : Stream, os : Stream) :

Ejecuta el *shell*.

Recibe dos flujos como parametro: un flujo de entrada y un flujo de salida, que sirven como medio de interacción del usuario físico con el sistema.

Esta generalización permitiría invocar *shells* que reciban órdenes de un archivo de entrada y devuelvan los resultados en otro archivo.

Internamente contiene el procesador de órdenes. Lee ordenes del flujo de entrada y realiza las acciones adecuadas invocando al resto de métodos del *shell* que manipulan físicamente el directorio. En cada orden se pasa como parámetro el directorio de trabajo para tratar los casos de manipulación del directorio personal o del directorio público.

Dir (os : Stream) :

Lista el contenido de un directorio.

Muestra por pantalla los *slots* de un directorio.

Recibe como parámetro un directorio porque puede ser que el usuario quiera ver el directorio publico o su propio directorio.

AddSlot (name : String, clase : String, rd : TDirectory) :

Añade un *slot* al directorio pasado como parámetro.

Crea un nuevo *slot* y construye una instancia de la clase solicitada, insertando el elemento en el directorio.

DelSlot (name : String, rd : TDirectory) :

Borra un *slot* del directorio.

Localiza el nombre del *slot* en el directorio, lo borra y borra físicamente la instancia asociada al *slot*.

RenSlot (name : String, name2 : String, rd : TDirectory) :

Cambia el nombre de un *slot*.

View (name : String, rd : TDirectory) :

Muestra el *slot* del sistema de ficheros solicitado por pantalla.

Sirve como guía para que el usuario conozca los métodos que tiene la referencia, qué parámetros y qué permisos de ejecución tiene.

FExec (name : String, metodo : String, rd : TDirectory) :

Prohibe la ejecución del método del *slot* solicitado.

ExecSlot (name : String, metodo : String, rd : TDirectory) :

Ejecuta el método del *slot* solicitado.

Si la ejecución del método está prohibida se atrapa la excepción y se devuelve un error al usuario.

MakePublic (name : String) :

Hace público un *slot*.

Crea un *slot* en el sistema de ficheros público con el mismo nombre pasado por parámetro. No se crea una nueva instancia sino que simplemente se asignan las referencias, por tanto si la instancia se borra esta no será accesible.

19 Aplicación a otros Sistemas y Trabajo Relacionado

19.1 Aplicación a la plataforma Java

Este tipo de protección para un sistema integral orientado a objetos, basada en capacidades, que combina la protección con el modelo de objetos del sistema al transformar las referencias en capacidades que contienen la información de protección, puede ser utilizado en sistemas parecidos. Un candidato obvio es la máquina virtual de Java [LY97] (JVM, *Java Virtual Machine*). Dotar a la máquina virtual de Java con este tipo de capacidades haría posible que la plataforma dispusiera de un mecanismo de protección de grano fino. Así se podrían construir políticas de seguridad más flexibles que las que se acostumbran a tener con modelos basados en variantes de la caja de arena.

La arquitectura de seguridad para la distribución 1.2 de la plataforma Java [GS98] (JDK1.2) mejora el modelo de la caja de arena sin romper la máquina virtual implementando una serie de abstracciones con clases básicas de la distribución. Se consigue una mayor finura en el control de acceso mediante una combinación de permisos de acceso (similares a capacidades), autoridades (*principals*, direcciones origen del código y firmas digitales de código) y dominios de protección (agrupan un conjunto de permisos de acceso determinados). Las clases (y por extensión sus instancias) se asignan a un dominio de protección mediante una política definida por el usuario, en función de la autoridad a que pertenezca la clase y su origen. El acceso de una clase está limitado al conjunto de recursos que defina su dominio de protección. En esencia se permiten definir varias cajas de arena (en lugar de una sola) particularizadas para conjuntos de clases.

Como ya se mencionó en el capítulo 8, este modelo, aunque mejora los modelos anteriores, sufre de muchas desventajas aún. Por ejemplo no permite seguir totalmente el principio de mínimo privilegio ni especificar permisos de acceso para objetos individuales, sólo para clases. Otro problema está derivado de la complejidad del mecanismo, que parece mayor de la necesaria. Todas las abstracciones están en alguna forma entremezcladas entre sí, y sólo tienen sentido cuando se utilizan todas en combinación. Esto produce un mecanismo grande y pesado, que hace que la base de computación fiable tenga que ser demasiado grande y no se pueda adaptar a entornos que no necesiten tantos requisitos de protección, como sistemas empotrados.

Un mecanismo de protección básico integrado en el modelo de objetos del sistema es más fácil de comprender, menos pesado, y por tanto menos proclive a la existencia de agujeros de seguridad en la implementación. Con las capacidades integradas en el modelo, los problemas anteriores que tiene el nuevo modelo de protección de Java no se presentan. Las abstracciones de alto nivel para la seguridad, como autoridades, etc. se pueden construir por encima de este mecanismo básico si se desea (como se ve en el ejemplo del capítulo 18).

El nuevo sistema de seguridad para la plataforma Java está más enfocado en la protección de recursos tradicionales del sistema, como ficheros, contra objetos no fiables descargados de la red que en el caso general de proteger los objetos entre sí.

Además, se presentan problemas de movilidad. El dominio tiene el conjunto de permisos común para un conjunto de objetos (por ejemplo todos los que vienen de la misma URL), pero para un objeto concreto puede que estos permisos no se utilicen. Esto restringe la movilidad de objetos individuales, puesto que la información de protección de un objeto se almacena en una entidad separada, el dominio. La migración de un objeto necesita de la migración de su dominio, que potencialmente puede ser muy grande, ya que contiene muchos más permisos de los que el objeto necesita estrictamente. Las capacidades son mucho mejores con respecto a la movilidad, puesto que cada objeto mantiene únicamente las capacidades que necesita, y son parte de su propia estructura y no se almacenan en otra entidad. Para migrar un objeto, únicamente hay que mover su estado, que ya contiene toda la información de protección que necesita.

19.2 Trabajo relacionado

El trabajo relacionado en el área de la aplicación de capacidades se presenta en forma de proyectos que pretenden introducir la protección basada en capacidades a entornos ya existentes (especialmente la plataforma Java) para construir sistemas distribuidos flexibles. El modelo concreto de capacidades varía.

19.2.1 Capacidades ocultas

Las capacidades ocultas [Hag96] son una aproximación en la cual los objetos de usuario están separados de las tareas de protección. Las capacidades son gestionadas por objetos externos que dan a los objetos de usuario las capacidades (referencias normales) de acuerdo con una política determinada. De esta manera los objetos se pueden reutilizar con varias políticas diferentes. Existen propuestas para incorporar este modelo a las plataformas CORBA y Java.

Reutilización de los objetos con diferentes políticas de seguridad

Esta aproximación también puede lograrse en el sistema integral, puesto que las aplicaciones no tienen por qué conocer la existencia de las capacidades y las pueden usar como simples referencias. Objetos externos pueden implementar las políticas pasando las capacidades apropiadas a estos objetos según las necesidades. Evidentemente, estos objetos que gestionan las capacidades deben conocer la existencia de capacidades y de la protección y utilizarían las posibilidades que ofrece la máquina para gestionar la información de protección de las capacidades (como la instrucción de restricción de permisos). Sin embargo, los objetos normales de usuario no realizarían estas operaciones y podrían ser reutilizados con diferentes políticas de seguridad sin hacer cambios.

19.2.2 El proyecto SLK y el J-Kernel

El proyecto SLK [HCC+97] pretende introducir las denominadas capacidades de tipo (*type-capabilities*) en la plataforma Java. Las capacidades de tipo ofrecen una combinación de comprobación de protección estática en tiempo de enlace, basada en un conjunto de permisos de acceso comunes para todos los objetos del mismo tipo, con una protección individual más normal basada en capacidades. La técnica de análisis estático apuntada en el capítulo 17 es similar a este concepto de protección en tiempo de enlace en el sentido de que intenta encontrar lugares en los que la protección pueda comprobarse de antemano una única vez.

Este proyecto ha evolucionado al J-Kernel [HCC+98], que distingue entre referencias locales y objetos capacidad. Los cargadores de clases generan representantes (*stubs*) que reescriben el código original, impidiendo el libre intercambio de referencias locales entre los dominios de protección, asegurando así que las capacidades se utilizan en la comunicación

entre dominios. Estos mecanismos en tiempo de ejecución se combinan con bibliotecas de utilidad en Java que implementan estas abstracciones: capacidades, etc.

Una funcionalidad similar existe en el sistema propuesto en este trabajo. La utilización de referencias locales puede asimilarse a la parte de las llamadas que no necesitan protección en una aplicación. Estas simplemente tendrían activado el permiso de salto de protección en la capacidad con lo que la velocidad de ejecución sería mayor. Los objetos capacidad se utilizan en las llamadas entre dominios, es decir, en las llamadas que necesitan protección. En estos casos se utilizan las capacidades con los permisos restringidos de manera adecuada. Sin embargo, no es necesario utilizar dos abstracciones diferentes (referencias normales y objetos capacidad) ni añadir bibliotecas adicionales al sistema.

En caso de que se desease además controlar el intercambio de referencias entre los dominios (control de la propagación), podrían utilizarse las técnicas descritas en el capítulo 9. Por ejemplo implantando un monitor de referencia aprovechando las posibilidades reflectivas de la máquina, o bien utilizando la técnica de representantes (*stubs*) que reescriben el código para impedir estos intercambios.

19.2.3 Problemas de la implementación mediante capas añadidas

Una característica común de estos diferentes tipos de capacidades, es que la implementación se realiza fundamentalmente mediante bibliotecas de usuario, gestores de objetos y mecanismos en tiempo de ejecución añadidos por encima de las plataformas existentes. Por tanto, la utilización de la mayor parte de la funcionalidad de las capacidades es mediante una limitación autoimpuesta a la hora de la programación, que no es parte del modelo de objetos ni de la infraestructura del sistema. Esto produce los ya mencionados problemas de aceptación por parte de los usuarios, mayores posibilidades de existencia de agujeros de seguridad en la implantación del sistema, adaptación a sistemas con requisitos diferentes, etc., que se evitarían utilizando un modelo de capacidades integrado en el modelo de objetos e implantado como parte del núcleo del sistema.

19.2.4 Flexibilidad de las capacidades orientadas a objetos

Un ejemplo más de la flexibilidad del modelo de capacidades propuesto en este trabajo puede verse en la facilidad con que se pueden desarrollar con el modelo las funcionalidades que proporcionan otras arquitecturas descritas en la parte del trabajo relacionado. Tanto la reutilización de objetos con diferentes políticas de seguridad de las capacidades ocultas, como la separación en referencias locales y llamadas con protección del J-Kernel encuentran acomodo dentro del sistema integral con el mecanismo de protección propuesto. Sin embargo, no se arrastran por ello problemas adicionales que presentan estos sistemas ni se obliga a la utilización de estas arquitecturas en sistemas que no las necesiten.

20 Conclusiones

20.1 Capacidades Orientadas a Objetos

Al inicio de este trabajo se mostró la necesidad de la aparición de Sistemas Integrales Orientados a Objetos (SIOO), como solución a los problemas que plantea la incorporación de las tecnologías orientadas a objetos en los sistemas actuales. Así mismo, se motivó la importancia de la protección para estos entornos y las carencias de los sistemas actuales que conducen a la creación de un nuevo modelo de protección del control de acceso para un SIOO.

Posteriormente se dedicó el trabajo a desarrollar un nuevo modelo de protección y a verificar la factibilidad del mismo. Para ello se estudiaron sistemas existentes, se establecieron tanto los requisitos generales para el modelo como los específicos para un SIOO, se diseñó el modelo y se analizaron sus ventajas. Por último, se realizó la implementación de un prototipo de SIOO incorporando el mecanismo de protección propuesto y se estudió el rendimiento del sistema. Adicionalmente se incluyeron ejemplos de las propiedades del sistema y la posible aplicación a sistemas existentes. El resultado es un mecanismo de protección para un SIOO sencillo, uniforme y homogéneo, más flexible y adaptable que otros sistemas existentes y con un rendimiento adecuado, que cumple los requisitos generales para un sistema de seguridad, así como los específicos para un SIOO.

En el capítulo 3 se mostraron los requisitos generales que debe cumplir un sistema de seguridad: mínimo privilegio, ahorro de mecanismos, aceptación, mediación total y diseño abierto.

En el capítulo 4 se estudiaron los diferentes modelos de seguridad existentes en la literatura. En el capítulo 5 se resume la arquitectura del SIOO sobre el que se va a desarrollar el nuevo mecanismo de protección, para en el siguiente capítulo establecer los requisitos específicos de diseño que tiene que cumplir el mecanismo: flexibilidad, movilidad, uniformidad en la orientación a objetos, homogeneidad y protección de granularidad fina.

En los siguientes capítulos se revisaron los mecanismos de protección de diferentes sistemas operativos, incluyendo a la plataforma Java, para pasar a elegir en el capítulo 9 a las capacidades como fundamento inicial mecanismo de protección por los problemas que presentan las listas de control de acceso y no presentan las capacidades: incumplimiento de los requisitos generales de seguridad (mínimo privilegio, etc.), sobrecarga, falta de escalabilidad, etc. El siguiente capítulo se dedica a estudiar el modelo de las capacidades con más profundidad.

El siguiente capítulo (11) se dedica a exponer el modelo de protección diseñado, que se ha denominado "capacidades orientadas a objetos". El modelo se basa fundamentalmente en:

- Fusión las referencias de la máquina abstracta con la información de protección de las capacidades
- Número de permisos variable dependiendo del objeto a proteger.
- Permiso especial "salto de protección" para acelerar el rendimiento del sistema.
- Operación de restricción de un permiso de acceso a un método dentro de una capacidad, que es la única operación adicional que hay que incluir.

En el capítulo 12 se resumen las ventajas del mecanismo: cumplimiento de todos los requisitos generales y específicos planteados, solución de los problemas comunes de un sistema de protección, y unas ventajas adicionales (protección automática de las capacidades y semántica completa de objetos mediante permisos de longitud variable). A continuación se dan ejemplos de diferentes políticas de seguridad que se podrían desarrollar a partir del mecanismo básico, para dar una idea de su flexibilidad.

En los siguientes capítulos se describe la implementación de prototipos para demostrar la factibilidad del sistema. En los capítulos 14 y 15 se resume la especificación, diseño e implementación del prototipo de máquina abstracta del SIOO en el que se introducirá el mecanismo de protección. El siguiente capítulo describe las modificaciones que se necesitan en el prototipo anterior para implantar el mecanismo en el sistema, desarrollándose al mismo tiempo una aplicación de la reflectividad que permite controlar el funcionamiento interno de la máquina desde código de usuario.

El capítulo 17 muestra como el rendimiento del sistema desarrollado es más que adecuado en aplicaciones reales, lo que evita el rechazo de los usuarios y la adopción sin inconvenientes del mismo. Además se implementó un sencillo entorno de usuario como ejemplo de la flexibilidad del mecanismo de protección y del sistema en su conjunto, que da una idea de que puede desarrollarse cualquier tipo de sistema que se pueda imaginar. El entorno permite la existencia de usuarios, les proporciona un sencillo *shell* para trabajar con el sistema. Además se incluye una política de seguridad que permite compartir documentos entre los usuarios, construida sobre el mecanismo de protección descrito.

Los diferentes resultados de este trabajo podrían aplicarse a sistemas similares. El capítulo 19 da un ejemplo de ello, con propuestas para mejorar la plataforma Java mediante la integración de capacidades en las referencias. También se muestra cómo el modelo propuesto puede sustituir y/o complementar ventajosamente otras propuestas de integración de capacidades en sistemas existentes, como por ejemplo a las capacidades ocultas (separación del código de protección del código funcional normal) o al J-Kernel (optimización de llamadas locales que no necesitan protección, usando el permiso "salto de protección").

20.2 Algunos resultados destacables

A continuación se resumen algunos resultados que se pueden destacar, agrupados en tres partes fundamentales.

20.2.1 Modelo de capacidades el más adecuado para un SIOO

20.2.1.1 Capacidades más adecuadas que listas de control de acceso

Entre los modelos de seguridad existentes, el más adecuado para fundamentar el mecanismo de protección de un Sistema Integral Orientado a Objetos (y en general de otro tipo de sistemas) es el modelo de capacidades. El modelo más utilizado en los sistemas actuales, el modelo de listas de control de acceso, tiene grandes deficiencias en múltiples áreas, que no presentan los sistemas basados en capacidades. El incumplimiento de principios básicos de diseño de mecanismos de seguridad, como el principio de mínimo privilegio, la falta de uniformidad y de escalabilidad, falta de granularidad fina, etc.

20.2.1.2 Peligro de las listas de control de acceso

Como efecto lateral, se descubrió el peligro de la utilización de listas de control de acceso como mecanismo de protección. Aunque su amplia utilización induce a los usuarios a creer que funcionan perfectamente y están protegidos, en realidad esto no es así. Ejemplos como la ilusión de revocación selectiva con listas de control de acceso, o la ilusión de la solución del problema de la propagación lo demuestran. El usuario cree que está protegido y que, por ejemplo, cuando cede un permiso a otro usuario, este no lo puede propagar y por tanto tiene la garantía de que este va a ser el único que pueda acceder. La realidad es que mediante una sencilla indirección, este segundo usuario puede propagar el acceso a quién desee. El peligro es mayor que si el mecanismo no pudiera proteger ese caso en concreto, puesto que el usuario cree que sí está protegido, cuando en realidad no lo está.

20.2.2 Modelo propuesto de capacidades orientadas a objetos: ventajas adicionales

A continuación se resumen algunas ventajas del modelo propuesto, derivadas fundamentalmente de la decisión básica de fundir las referencias de la máquina abstracta con la información de protección de las capacidades.

20.2.2.1 Cumplimiento de todos los requisitos de seguridad en el diseño de un mecanismo de protección

El modelo cumple todos los requisitos exigidos en general a un mecanismo de protección, y específicamente para un SIOO (capítulo 12), lo cual no es alcanzado por otros modelos existentes en los sistemas examinados.

20.2.2.2 Uniformidad

Se mantiene totalmente la uniformidad en la orientación a objetos del sistema. La inclusión del mecanismo de protección no afecta al modelo de objetos del sistema de ninguna manera, manteniéndose totalmente la semántica anterior del sistema. Simplemente se añade una nueva operación para restringir los permisos de una capacidad.

20.2.2.3 Homogeneidad

Sólo es necesario este mecanismo de protección en el sistema. Se protegen todos los objetos por igual (homogeneidad), con lo que no es necesario utilizar (ni aprender) diferentes mecanismos de protección, ni de inventar nuevos mecanismos *ad-hoc* con posterioridad para proteger nuevos recursos que puedan aparecer (puesto que se tratarán como objetos en el sistema integral).

20.2.2.4 Flexibilidad

El mecanismo es suficientemente flexible para acomodar diferentes políticas de seguridad que se puedan construir a partir de éste, como se muestra en el capítulo 13 definiendo algunas políticas de ejemplo, y en el capítulo 18, en el que se implementa un entorno que incorpora una.

20.2.2.5 Adaptabilidad

El mecanismo es lo suficientemente pequeño (aunque potente a la vez) para aportar las ventajas de la protección a cualquier entorno en que se aplique el sistema integral, sin obligar a pagar un precio adicional (en sobrecarga de ejecución y tamaño del código) cuando no se necesite seguridad. Por ejemplo, un sistema empotrado utilizaría el mecanismo básico sin más. Sistemas más sofisticados, como un sistema de usuario final de escritorio, podrían añadir conceptos adicionales, como usuarios, etc., pero éstos sólo se construirían si fuera necesario, no es necesario pagar el precio que conllevan si no van a utilizarse. Un ejemplo es el sencillo entorno desarrollado en el capítulo 18.

20.2.2.6 Sencillez de implementación y conceptual (aumento de la productividad)

El mecanismo es conceptualmente muy sencillo y se integra de manera continua con el modelo de objetos. Esto hace que sea fácilmente entendible por los usuarios, y que éstos lo utilicen. Además, al ser el único mecanismo que se utiliza (homogeneidad) no necesitan aprender nuevos conceptos, con lo que aumenta la productividad.

Al ser tan sencillo, también lo es la implementación (como se ve en el capítulo 16), con lo que la posibilidad de introducir agujeros de seguridad se reduce.

20.2.2.7 Rendimiento adecuado de la protección basada en capacidades

El rendimiento del mecanismo parece más que adecuado en aplicaciones reales (como se muestra en el capítulo 17), gracias a la existencia del permiso de "salto de protección" que evita la sobrecarga de comprobación de permisos cuando no es necesaria. La sobrecarga de la protección en un sistema basado en capacidades de este estilo no es obstáculo para la adopción del mismo por los usuarios, si no todo lo contrario, por lo pequeña de esta para aplicaciones reales, teniendo en cuenta los beneficios que aporta.

20.2.2.8 Permisos de longitud variable. Semántica de objetos completa

Al contrario que otros sistemas que utilizan capacidades, que sólo protegen un número fijo de operaciones de los objetos, las capacidades del modelo propuesto tienen un conjunto de permisos de longitud variable, dependiendo del objeto al que se haga referencia. Esto hace posible mantener la semántica completa del modelo de objetos, en la que un objeto puede tener un número arbitrario de métodos, sin un máximo fijo determinado. El alto nivel de la máquina abstracta, en la que las capacidades son una abstracción más que las aplicaciones no ven con un tamaño físico determinado permite este tipo de capacidades.

20.2.2.9 Protección automática de capacidades

Las ventajas de las capacidades segregadas y de las capacidades dispersas se combinan, evitando sus inconvenientes. Al asegurar la máquina abstracta que las capacidades sólo pueden manipularse a través de las operaciones definidas para ello, se garantiza que nunca se podrá acceder directamente a su representación y cambiarla arbitrariamente (la falsificación es imposible). Por otro lado, su condición de referencias permite incluirlas en las estructuras normales de usuario y que estos las intercambien libremente, lo que es más flexible.

20.2.3 Aplicación a otros sistemas

20.2.3.1 Insuficiencia de los mecanismos de protección de la plataforma Java

El análisis de los diferentes mecanismos de seguridad de la plataforma Java reveló (capítulo 8) que, a pesar de la mejora efectuada en los mismos, aún son insuficientes puesto que no alcanzan niveles óptimos en algunos aspectos: principio del mínimo privilegio, mediación total, sencillez, falta de uniformidad, flexibilidad y adaptabilidad, etc.

20.2.3.2 Mejora de la plataforma Java

La aplicación análoga del modelo de protección propuesto aquí (o de la mayor parte de la filosofía de diseño del mismo) permitiría dotar a la plataforma Java de alguna de las ventajas del modelo, y mejorar el nivel de seguridad de la misma en parte de los aspectos descritos anteriormente.

20.2.4 Implementación de prototipos

A continuación se muestran algunos resultados derivados de la experiencia acumulada durante la implementación de los prototipos de ejemplo.

20.2.4.1 Flexibilidad del concepto de SIOO

El propio concepto de Sistema Integral Orientado a Objetos demostró su flexibilidad, puesto que se pudo desarrollar fácilmente un entorno de computación (capítulo 18) para usuarios finales (que sería parte de la funcionalidad de un sistema operativo) sobre la máquina abstracta ofrecida por el sistema. El propio entorno se describe como un conjunto de clases que el usuario puede instanciar como cualquier otra clase en el sistema. Por ejemplo, una instancia de la clase Directorio puede albergar instancias de cualquier otra clase, como instancias de Directorio a su vez, o de Shell, o incluso del entorno completo de usuario.

20.2.4.2 Importancia de la reflectividad para la extensibilidad

La reflectividad se muestra como un elemento muy importante para conseguir una extensibilidad sencilla y uniforme del sistema. Un ejemplo es la introducción de la clase reflectiva System, que representa funcionalidad de la máquina abstracta (por ejemplo tiene métodos para controlar la carga y descarga de clases en el área de instancias). El sistema crea inicialmente una instancia de esta clase, que puede ser utilizada por un usuario normalmente como cualquier otro objeto. De esta manera se logra extender el sistema mediante clases de usuario que pueden interactuar con la máquina que las soporta para ampliar las propiedades del sistema, manteniendo en todo momento la uniformidad en la orientación a objetos.

Por supuesto, el acceso a esta instancia puede protegerse como el acceso a cualquier otro objeto mediante la restricción de permisos que proporciona el mecanismo de protección. Este es un ejemplo más de la uniformidad y homogeneidad que tiene el sistema con su mecanismo de protección, y que la reflectividad permite aplicar incluso en la propia máquina que soporta el sistema.

20.3 Trabajo y líneas de investigación futuras

20.3.1 Estudio de patrones de comportamiento de aplicaciones

Debido a la juventud del sistema integral, aún no han sido desarrolladas muchas aplicaciones de gran envergadura. Es necesario construir estas aplicaciones, especialmente aplicaciones representativas de aplicaciones comunes en la práctica, para analizar el comportamiento de las mismas en cuanto al uso de la seguridad y la protección. En concreto es importante conocer el número de llamadas que necesitan protección dentro de la aplicación. Esta información permitirá afinar mejor la implementación del mecanismo de protección, así como para evaluar la posible necesidad de introducir cambios en el diseño.

20.3.2 Desarrollo completo de un entorno de usuario

Otra parte, relacionada con el desarrollo del sistema operativo, es la construcción de un entorno de usuario más completo que el sencillo prototipo implementado para mostrar el funcionamiento del mecanismo de protección. Se trataría de dar al entorno una funcionalidad similar a la existente en entornos de sistemas operativos tradicionales como Unix, al menos inicialmente.

Por otro lado, y ya más directamente relacionado con el sistema de seguridad, es necesario desarrollar también políticas de seguridad más complejas que la introducida en el prototipo, que se ajusten a las necesidades concretas de aplicaciones reales. Además, es otra manera de verificar aún más la posibilidad de implantación de diferentes políticas sobre el mecanismo básico.

20.3.3 Mejora en la implementación de la máquina

La introducción de optimizaciones en la implementación interna de la máquina es un campo muy amplio. Un aspecto general es el de la optimización del funcionamiento de la máquina en general, por ejemplo usando técnicas de compilación dinámica. Relacionado directamente con la protección se encuentra la parte de optimización de todo lo involucrado con el funcionamiento del mecanismo de protección. Por ejemplo la implementación física del vector de permisos, la consulta de los métodos que tiene un objeto, la secuencia de llamadas internas para la comprobación de permisos, etc.

Otro apartado interesante es la investigación de las técnicas de comprobación estática de permisos que se apuntaron en el capítulo 17.

20.3.4 Reflectividad: modelo reflectivo para soporte de monitores de referencia

Otra línea de trabajo está relacionada con la reflectividad. Por una parte se puede profundizar en la adición reflectiva de más funcionalidad de la máquina accesible al espacio de usuario, en general. Se trataría de seguir con la línea iniciada con algunos métodos incluidos en la nueva clase System de la máquina.

El estudio del mejor modelo reflectivo a exponer al usuario para permitir la implementación reflectiva de monitores de referencia que controlen la propagación, delegación y/o revocación de capacidades es otro aspecto interesante.

20.3.5 Aplicación a la plataforma Java

Como se apuntó en el capítulo 19, es posible aplicar gran parte de la filosofía y conceptos desarrollados para el mecanismo de protección basado en capacidades a la plataforma Java o a otros sistemas similares. Se trataría de estudiar la mejor manera de introducir la mayor parte de esta funcionalidad en la plataforma, del mejor modo posible.

BIBLIOGRAFÍA

[Álv96a] Fernando Álvarez García. "Distribución en sistemas operativos orientados a objetos". *II Jornadas sobre Tecnologías Orientadas a Objetos*. Oviedo. Marzo de 1996.

[Álv96b] Darío Álvarez Gutiérrez. "Persistencia en un sistema operativo orientado a objetos". *II Jornadas sobre Tecnologías Orientadas a Objetos*. Oviedo. Marzo de 1996.

[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 de la Universidad de Oviedo. Marzo 1998.

[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.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall. 1986.

[BL73] D.E. Bell y L.J. LaPadula. *Secure Computer Systems: Mathematical Fundations.* ESD-TR-278,1, ESD/AFSC, Hadscom AFB, Bedford, MA., EE.UU. 1973.

[Boo91] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings. 1991.

[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.

[BRJ96] Grady Booch, James Rumbaugh e Ivar Jacobson. *The Unified Modeling Language for Object-Oriented Development, Version 0.9*. Rational Software Corporation. Julio de 1996.

[Cah96] Vinny Cahill. *Flexibility in Object-Oriented Operating Systems: A Review*. Technical Report TCD-CS-96-05. Trinity College Dublin. 1996.

[CBT+92] V. Cahill, S. Baker, B. Tangney, C. Horn y N. Harris. On Objecto Orientation as a Paradigm for General Purpose Distributed Operating System. Proceedings of the *ACM SIGOPS*. 1992.

[CIA96] Juan Manuel Cueva Lovelle, Raúl Izquierdo Castanedo y Darío Álvarez Gutiérrez. "Oviedo3: Acercando las tecnologías orientadas a objetos al hardware". *I Jornadas de trabajo en Ingeniería de Software*. Sevilla. Noviembre de 1996.

[CIM+93] R. Campbell, N. Islam, P. Madany y D. Raila. "Designing and Implementing Choices: an Object-Oriented System in C++". *Communications of the ACM*. Septiembre de 1993.

[CLF+93] J. S. Chase, H.M. Levy, M.J. Feeley y E.D. Lazowska. *Sharing and protection in a single address space operating system*. Technical Report 93-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, EE.UU. Abril de 1993.

[CM90] R.H. Campbell y P.W. Madany. "Considerations of Persistence and Security in Choices, an Object-Oriented Operating System", Proceedings of the *International Workshop on Computing Architectures to Support Security and Persistence of Information*. Mayo de 1990.

[CSC72] F. Corbató, L. Saltzer y C. Clingen. "Multics. The first seven years". *AFIPS Conference* proceedings, vol 40. 1972

[Cue95] Juan Manuel Cueva Lovelle. *Conceptos básicos de traductores, compiladores e intérpretes, quinta edición*. Colección Cuadernos Didácticos del Departamento de Matemáticas de la Universidad de Oviedo. 1995.

[DAG98] María Ángeles Díaz Fondón, Darío Álvarez Gutiérrez, Armando García Mendoza Sánchez, Fernando Álvarez García, Lourdes Tajes Martínez y Juan Manuel Cueva Lovelle. "Merging Capabilities with the Object Model of an Object-Oriented Abstract Machine". *ECOOP Workshop on Distributed Security.* Bruselas, Bélgica, 1998.

[DAG99] María Ángeles Díaz Fondón, Darío Álvarez Gutiérrez, Armando García Mendoza Sánchez, Fernando Álvarez García, Lourdes Tajes Martínez y Juan Manuel Cueva Lovelle. "Integrating Capabílities into the Object Model to Protect Distributed Object Systems". *International Symposium on Distributed Objects and Applications (DOA'99)*. Edimburgo, Escocia, Reino Unido. 1999.

[DAT98] María Ángeles Díaz Fondón, Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García y Juan Manuel Cueva Lovelle."Capability-Based Protection for Integral Object-Oriented Systems". Proceedings of the *22 Conference in Computer Software and Applications (COMPSAC'98)*. IEEE Computer Society Press. 1998. Pág. 344-349.

[DBF+94] A. Dearle, R. di Bona, J. Farrow, F. Henskens, D. Hulse, A. Lindstrom, S. Norris, J. Rosemberg y F. Vaughan. "Protection in the Grasshopper Operating System". *6th International Workshop on Persistent Object Systems*, Tarascon, Francia. Septiembre de 1994.

[Dei93] H.M.Deitel. *Sistemas Operativos, Segunda Edición.* Addison-Wesley. 1993.

[Den76] D.E. Denning. “A Lattice Model of Secure Information Flow”.*Communications of the ACM*, vol 5. 1976.

[DFW96] Dean Drew, Edward W. Felten y Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. Proceedings of the *1996 IEEE Symposium on Security and Privacy*. Oakland, California, EE.UU. Mayo de 1996.

[Día96] María Ángeles Díaz Fondón. “Seguridad en un sistema operativo orientado a objetos”. *II Jornadas sobre Tecnologías Orientadas a Objetos.* Oviedo. Marzo de 1996.

[DoD83] Department of Defense. *Trusted Computer System Evaluation Criteria.* CSC-STD-001-83. Computer Security Center of the Department of Defense, EE.UU. 1983.

[Eri99] ERights.org. *Prohibiting Delegation*. 1999.

URL: http://www.erights.org/elib/capability/delegations.html.

[Fab74] R.S. Fabry. “Capability-based addressing”. *Communications of the ACM,* vol 17. 1974. Pág. 403-411.

[Fer99] L.C. Fernández Lozano. *Reflectividad estructural para Carbayonia*. Proyecto Fin de Carrera. Escuela Universitaria de Ingeniería Técnica en Informática de Gijón de la Universidad de Oviedo. Septiembre de 1999.

[Fla97] D. Flanagan. *"JavaScript: The Definitive Guide, 2nd ed"*. O'Reilly and Associates, Inc. 1997

[Gar99] Armando García-Mendoza Sánchez. *Implantación de un mecanismo uniforme de protección en el Sistema Integral Orientado a Objetos Oviedo3*. Proyecto Fin de Carrera 982021. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Noviembre de 1999.

[GD72] G.S. Graham y P.J.Denning. “Protection: Principles and Practices”. Proceedings of the *AFIPS Spring Joint Computer Conference*. 1972.

[GJS96] J. Gosling, B. Joy y G. Steele. *The Java Language Specification*. Addison-Wesley. 1996

[Gos91] Andrzej Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley. 1991.

[GR83] A. Goldberg y D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley. 1983.

[GS98] L. Gong y R. Schemers. “Implementing Protection Domains in the Java Development Kit 1.2”. *Int. Soc. Symp. on Network and Distributed System Security*. San Diego, EE.UU. 1998.

[GW96] Brendan Gowing y Vinny Cahill. “Reflection + Micro-Kernels: An Approach to Supporting Dynamically Adaptable System Services”. *CaberNet Radicals '96*. Connemara, Irlanda. 1996.

[Hag94] D. Hagimont. “Protection in the Guide object-oriented distributed system”. Proceedings of the *ECOOP'94*. 1994.

[Hag96] D. Hagimont, J. Mosière, X. Rousset de Pina y F. Saunier.. "Hidden Capabilities". *16th International Conference on Distributed Computing Systems*. Mayo de 1996.

[Har85] N. Hardy. "KeyKOS Architecture". *Operating Systems Review*, vol. 19 no.4. Octubre de 1985.

[Har88] Norm Hardy. "The Confused Deputy, or why capabilities might have been invented". *Operating Systems Review*. Octubre de 1988. Pág. 36-38.

[HCC+97] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu y T. von Eicken. *SLK: A Capability System Based on Safe Language Technology*. CS Department, Cornell University. March 1997.

URL http://www2.cs.cornell.edu/Slk/papers/slk.ps

[HCC+98] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu y T. Von Eicken "Implementing Multiple Protection Domains in Java". *1998 Usenix Annual Technical Conference*. New Orleans, Lousiana, EE.UU. Junio de 1998.

[HKM+96] D. Hagimont, S. Krakowiak, J. Mosière y X. Rousset de Pina. "A Selective Protection Scheme for the Java Environment".1996

[HKR92] D. Hagimont, S.Krakowiak and X. Rousset de Pina. "Protection in an object-oriented distributed virtal machine". *International Workshop on Object Oriented Operating System*. 1992.

[HMR+96] D. Hagimont, J. Mosière, X. Rousset de Pina y F. Saunier. "Hidden Capabilities". *16th International Conference on Distributed Computing Systems*. Mayo de 1996.

[HRU76] M.A.Harrison, W.L. Ruzzo, J.D. Ullman. "Protection in Operating Systems". *Communications of the ACM*. 1976.

[IJ62] J.K. Iliffe y J.G. Jodeit. "A Dynamic Storage Allocation System". *Computer Journal*, vol. 5, no. 3. 1962.

[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.

[Kim97] Emin Gün Sirer, Sean McDirmid y Brian Bershad. "Kimera Architecture for Java". 1997.

URL: http://kimera.cs.washington.edu/overview.html.

[KJS96] Douglas Kramer, Bill Joy y David Spenhoff. *The Java™ Platform White Paper*. JavaSoft. Mayo de 1996.

URL: ftp://ftp.javasoft.com/docs/JavaPlatform.ps

[Kow90] O.C. Kowalski y H. Härtig. *Protection in the Birlix Operating System.* Proceedings of the *International Conference on Distributed Computing Systems*. May 1990.

[Lam71] B.W. Lampson. "Protection". *Fifth Princenton Conference on Information and Systems Sciences*. 1971.

[Lam73] B.W. Lampson. "A Note of the Confinement Problem". *Communications of the ACM*, vol. 10. Octubre de 1973.

[Lan89] Charles R. Landau. "Security in a Secure Capability-Based System". *Operating Systems Review*. Octubre de 1989.

[Lan81] Carl E. Landwehr. "Formal Models for Computer Security". *ACM Computing Surveys*, vol. 13, no. 3. Septiembre de 1981.

[Lev84] H.M. Levy. *Capability-Based Computer Systems*. Digital Press. 1984.

[LY97] Tim Lindholm y Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley. 1997.

[Mae87] P. Maes. "Concepts and Experiments in Computational Reflection". En Proceedings of the *1987 OOPSLA*. 1987. Pág. 147-155.

[MC96] Microsoft Corporation. *Proposal for Authenticating Code Via the Internet*. 1996

URL:http://www.microsoft.com/security/tech/authcode/authcode-f.htm.

[Mil99] Mark Miller. Computer Security: "Fact Forum Framework". 1999.

URL: http://www.caplet.com/security/taxonomy/

[MOO87] M. Maekawa, A.E. Oldehoeft y R.R. Oldehoeft. *Operating Systems. Advanced Concepts.*Benjamin Cummings. 1987.

[MSC97] M. Miller, M. Svodoba, D. Crockford, C. Mornigstar, N. Hardy y M. Stiegler. *Introduction to Capability Based Security*. Electric Communities. 1997.

URL: http://www.communities.com/company/papers/security/index.html.

[NAJ+76] K.V. Nori, U. Ammann, K. Jensen, H.H. Nageli y C. Jacobi. "The Pascal-P Compiler: Implementation Notes". *Bericht 10, Eidgengössische Technische Hochschule*. Zurich, Suiza. Julio de 1976.

[NL96] G.C. Necula y P. Lee. "Safe kernel extensions without run-time checking". Proceedings of the *Second Symposium on Operating Systems Design and Implementation*. 1996.

[Nut92] Gary J. Nutt. *Centralized and Distributed Operating Systems.* Prentice Hall. 1994.

[RA90] J. Rosenberg y D. Abramson. "The MONADS Architecture: A Layered View". Proceedings of the *4th International Workshop on Persistent Object Systems*. 1990.

[RHB+86] S.A. Rajunas, N. Hardy, A.C. Bomberger, W.S. Frantz y C.R. Landau. "Security in KeyKOS". Proceedings of the *1986 IEEE Symposium on Security and Privacy.* Abril de 1986.

[Sch97] B. Schmitt. *Shockwave Studio: Designing Multimedia for the Web*. O'Reilly and Associates. 1997

[Shap99] Jonathan Strauss Shapiro. *Eros: a Capability System*. Tesis Doctoral. Universidad de Pennsylvania, EE.UU. 1999.

[Sil98] Abraham Silberschatz y Peter Baer Galvin. *Operating System Concepts.* Addison-Wesley. 1998.

[SS75] J.H. Saltzer y M.D. Schroeder. "The Protection of Information in Computer Systems". Proceedings *of the IEEE*. vol. 63, no. 9. Septiembre de 1975. Pág. 1278-1308.

[SS94] Mukesh Singhal y Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill. 1994.

[Stal97] William Stallings. *Operating Systems. Internals and Design Prniciples, Third Edition.* Addison-Wesley. 1997.

[SW97] J. S. Shapiro y S. Weber. *Verifying Operating System Security*. Department of Computer and Information Science Technical Report MS-CIS-97-26, University of Pennsylvania. 1997.

[TAA+97] Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Juan Manuel Cueva Lovelle, Darío Álvarez Gutiérrez y Raúl Izquierdo Castanedo. "Concurrencia en un sistema operativo orientado a objetos basado en una máquina abstracta reflectiva orientada a objetos". *VII Jornadas de Paralelismo*. Cáceres. Septiembre de 1997.

[Taj96] Lourdes Tajes Martínez. "Introducción de la concurrencia en sistemas operativos Orientados a Objetos". *II Jornadas sobre Tecnologías Orientadas a Objetos*. Oviedo. Marzo de 1996.

[Tan92] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[Tan98] A.S. Tanenbaum. *Sistemas Operativos, Diseño e Implementación, Segunda edición*. Prentice Hall 1998.

[TMV86] A.S. Tanenbaum, S.J. Mullender y R. Van Renesse. "Using Sparse Capabilities in a Distributed Operating System". Proceedings of the *6th International Conference on Distributed Computing Systems*. 1986.

[TW97] A.S. Tanenembaum y A.S. Woondhull. *Sistemas Operativos, Diseño e Implementación.* Prentice Hall, 1997.

[Unio00] Universidad de Oviedo. página Web. Enero de 2000.

URL: http://www.uniovi.es/

[VK83] V.L.Voydock y S.T. Kent. "Security Mechanisms in High-Level Network Protocols". *Computing Surveys*, vol. 15, no. 2. 1983.

[VRH93] J. Vochteloo., S. Rusell y G. Heiser. "Capability-based Protection in a Persistent Global Virtual Memory System". *International Workshop on Object Orientation in Operating Systems (IWOOOS'93).* 1993.

[VVG95] M. De Vivo, G.O. de Vivo y L. González. "A Brief Essay on Capabilities". *SIGPLAN Notices*, vol. 30 no. 7. Abril de 1995.

[W74] W. Wulf et al. HYDRA: The kernel of a multiprocessor Operating System. *Communications of the ACM*, vol 17. Junio de 1974.

[WAB+94] R. Wahbe, M. Abadi, M. Burrows y E. Wobber. Efficient software-based fault isolation. Proceedings of the *Fourteenth Symposium on Operating System Principles*. 1994.

[Wal99] Dan Seth Wallach. *A New Approach to Mobile Code Security*. Tesis Doctoral. Universidad de Princeton, EE.UU. Enero de 1999.

[WBD+97] D.S. Wallach, D. Balfanz, D. Dean y E.W. Felten. "Extensible Security Architectures for Java". *16th Symposium on Operating Systems Principles*. Saint-Malo, Francia. 1997.

[WLH81] W. Wulf, R. Levin y S.P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill. 1981.