ÍNDICE

1. Introducción 2
2. Fundamentos sobre procesos 4
2.1. Sincronización de procesos mediante exit() y wait() 5

1. INTRODUCCIÓN

Las aplicaciones distribuidas existen en el mundo de la informática en una u otra forma desde hace algún tiempo, aunque no se hayan llamado explícitamente sistemas distribuidos y no tuviesen la flexibilidad que tienen hoy en día.

La definición más comúnmente aceptada de aplicación o software distribuido es: "Un conjunto de procesos en la misma máquina o en máquinas distintas que cooperan entre sí para alcanzar un objetivo común". Como idea de partida, para que se de la cooperación es necesario que los procesos puedan comunicarse entre sí y sincronizar sus actuaciones.

Al principio sólo existían los grandes ordenadores o mainframes en los que se ejecutaban Sistemas de gestión de Base de Datos (SGBD) jerárquicos. La conexión de los usuarios con el ordenador central se hacía mediante terminales sin capacidad de procesamiento (dumb terminals) con conexiones punto a punto con el servidor. Los grandes ordenadores tenían un coste muy elevado y eran difíciles de mantener pero eran capaces de dar servicio a un número muy elevado de usuarios y tenían la ventaja (o desventaja) de ser administrados de forma centralizada. En este tipo de sistemas el software era monolítico, es decir, la interfaz de usuario, la lógica de negocio y el acceso a las bases de datos estaban contenidos en una misma aplicación que se ejecutaba en el mainframe. Dado que las terminales utilizadas para conectarse al ordenador central no tenían ninguna capacidad de proceso, la aplicación entera se ejecutaba completamente en el ordenador central.

La aparición de los PCs y el auge de las redes locales introdujeron un cambio muy importante en las arquitecturas monolíticas y las aplicaciones para ellas desarrolladas y es en esta época cuando surge el modelo cliente/servidor, el cual es el paradigma de computación distribuida más utilizado hoy día. Las entidades básicas de este modelo son los procesos servidores, los cuales gestionan recursos (ej.: las tablas de un SGBD), por otro lado están los procesos clientes los cuales necesitan acceder a los recursos gestionados por los primeros. Un proceso cliente solicita usar un recurso enviándole una petición al servidor, el cual procesa dicha petición y le envía una respuesta al cliente. Un proceso no tiene por qué comportarse exclusivamente como cliente o como servidor, sino que su rol depende del contexto, así un proceso puede ser servidor de otro pero a su vez, para servir la petición en curso puede necesitar de los servicios proporcionados por un tercer proceso. La mayor parte de las aplicaciones más populares que se utilizan hoy en día en Internet son aplicaciones cliente/servidor (ej.: telnet, ftp, e-mail, web, etc.). El desarrollo de este tipo de aplicaciones requiere que los implementadores tengan en cuenta los problemas inherentes de la computación distribuida derivados de la concurrencia de procesos.

Las aplicaciones basadas en el paradigma cliente/servidor permitieron que parte del procesamiento realizado en el servidor fuese descargado a los PCs cliente. Las aplicaciones cliente/servidor normalmente distribuyen los componentes de la aplicación de forma que la base de datos reside en el servidor, la interfaz de usuario reside en el cliente, y la lógica de negocio puede residir tanto en el cliente (en forma de código), como en el servidor (en forma de procedimientos almacenados) o incluso en ambos.

La arquitectura cliente/servidor fue, en algunos aspectos, una revolución que cambió la vieja forma en que se hacían las cosas. A pesar de resolver muchos de los problemas de las aplicaciones basadas en mainframes, la arquitectura cliente/servidor tenía sus propios problemas. Por ejemplo, como la lógica de negocio y el acceso a la base de datos estaban contenidos normalmente en la parte cliente, cualquier cambio de la lógica de negocio, en el acceso a la base de datos o en la propia base de datos necesitaría actualizar la parte cliente de todos los usuarios de la aplicación. Los problemas con esta tradicional arquitectura cliente/servidor, a menudo denominada cliente/servidor de dos capas fueron resueltos con lo

que se conoce con el nombre de cliente/servidor multicapa. Conceptualmente, una aplicación puede tener cualquier número de capas, pero las arquitecturas multicapa suelen tener sólo tres capas. Estas arquitecturas de tres capas dividen la arquitectura del sistema en tres capas lógicas: la capa de interfaz con el usuario, la capa de la lógica de negocio y la capa de acceso a la Base de Datos.

De esta forma, la parte cliente va a ser básicamente una interfaz, que no tiene por qué cambiar si cambian la lógica de negocio o la forma de acceder a los datos. Las arquitecturas cliente/servidor multicapa mejoran a las arquitecturas de dos capas en que, por un lado, hacen la aplicación más robusta al aislar al cliente de los cambios en el resto de la aplicación y por otro lado permiten una mayor flexibilidad en la utilización de la misma.

Aunque se pueden desarrollar aplicaciones distribuidas utilizando tecnologías de bajo nivel (mecanismos IPC, sockets, etc.) veremos que existen mecanismos de más alto nivel que abstraen al desarrollador de los aspectos relativos a la comunicación permitiéndole centrarse en los puramente funcionales. Se verá también cómo estos mecanismos de alto nivel se basan en los mecanismos de bajo nivel.

Un mecanismo comúnmente utilizado para implementar el modelo cliente/servidor son las llamadas a procedimientos remotos o RPC. Últimamente, dado que la orientación a objetos ha probado su validez para desarrollar y mantener grandes aplicaciones, se está aplicando el paradigma de la orientación a objetos en la construcción de aplicaciones distribuidos.

El siguiente paso en la evolución de la arquitectura de las aplicaciones distribuidas (de las cuales las tradicionales aplicaciones cliente/servidor no son mas que una parte) es el modelo de sistemas distribuidos orientados a objetos. Esta arquitectura lleva al concepto de cliente/servidor de múltiples capas hasta su conclusión natural. Más que diferenciar entre capa de interfaz, lógica de negocio y acceso a los datos, el modelo de sistemas distribuidos orientados a objetos simplemente lo que hace es exponer toda la funcionalidad de la aplicación como objetos, cada uno de los cuales puede hacer uso de alguno de los servicios proporcionados por otros objetos en el sistema o incluso en otros sistemas conectados con el nuestro a través de una red. La interfaz de un objeto está compuesta por el conjunto de métodos públicos incluyendo los parámetros de entrada/salida. La interfaz de un objeto especifica a otros objetos qué servicios oferta y cómo deben de utilizarse. Así como la interfaz de un objeto puede permanecer constante, la implementación de los métodos de la interfaz puede cambiar sin afectar a otros componentes del sistema que utilizan dicho interfaz.

Es un hecho que los ordenadores existentes en una red son normalmente heterogéneos, tanto en su arquitectura hardware como sistema operativo. Incluso los lenguajes empleados para desarrollar aplicaciones orientadas a objetos son distintos. Esto es lo que se denomina un entorno distribuido heterogéneo. Para permitir la comunicación entre objetos en un entorno tan heterogéneo necesitamos una compleja pieza de software adicional que haga de pegamento y que recibe el nombre de plataforma middleware.

Finalmente, el desarrollo de aplicaciones para la Web está actualmente en auge, siendo este entorno el entorno de programación distribuida por excelencia.

2. FUNDAMENTOS SOBRE PROCESOS

Los procesos en UNIX presentan en su contexto los elementos recogidos en la Figura 1. Cualquier proceso UNIX puede crear un nuevo proceso empleando la llamada al sistema **fork()**. Esta es una de las llamadas al sistema más simples, pero más difícil de comprender, no lleva parámetros y devuelve un entero.

Atributo/Recurso	Descripción
ID de proceso	Un valor entero único que identifica a cada proceso
ID del padre del proceso	El ID del proceso que ha creado al actual
ID del usuario real	El ID del usuario que inicio el proceso actual
ID del usuario efectivo	El ID del usuario cuyos permisos están siendo empleados en este proceso. Normalmente este ID es el mismo que el del usuario real.
Directorio actual	El directorio de punto de partida para la búsqueda de paths relativos
Tabla de descriptores de fichero	Una tabla que contiene los datos de todos los flujos de entrada/salida abiertos por el proceso. La tabla está indexada por un índice entero denominado descriptor de fichero.
El entorno	Una lista de strings de la forma VARIABLE=VALOR, empleadas para parametrizar el comportamiento de ciertos programas. El entorno se puede obtener llamando a la rutina de librería **getenv()**.
Espacio de código	La parte de la memoria donde se almacena el código.
Espacio de datos	La parte de la memoria donde se almacenan las variables globales y otros datos estáticos.
Pila	La parte de la memoria empleada para almacenar las variables locales (que se encuentran dentro de las funciones)
Heap	La parte de la memoria reservada dinámicamente en respuesta a llamadas explícitas a las funciones de librería **malloc()** y **free()**.
Prioridad	Parámetros que controlan la planificación de procesos.
Señales	Máscaras que indican que señales están en espera, cuales están bloqueadas, etc.
Umask	Máscara cuyo valor asegura que los permisos de acceso que especifica no estén activados cuando este proceso crea un fichero.

Figura 1. Tabla de atributos y recursos de un proceso

La llamada a **fork()** crea un nuevo proceso que es exactamente un copia del original, donde se heredan/comparten ciertos componentes del proceso original y se copian otros. (ver Figura 2). El proceso original se denomina padre y al nuevo se le denomina hijo. El caso es que cuando se crea un proceso hijo es porque se desea que ese hijo haga algo distinto del padre. Por lo que debe existir una forma de diferenciarlos, que es el valor devuelto por la función **fork()**: el valor devuelto en el proceso padre es el ID del hijo. En el proceso hijo **fork()** devuelve 0. Por esto la llamada a **fork()** aparecerá invariablemente dentro de un if de modo que cada proceso tome una de las ramas del if.

Atributo/Recurso	¿Se hereda?
ID de proceso	No
ID del usuario real	Si
ID del usuario efectivo	Si
Directorio actual	Si

Atributo/Recurso	¿Se hereda?
Tabla de descriptores de fichero abiertos	Copiada; los descriptores de ficheros son compartidos.
El entorno	Si
Espacio de código	Compartido
Espacio de datos	Copiado
Pila	Copiado
Heap	Copiado
Comportamiento del manejo de señales	Copiado

Figura 2. Tabla de atributos heredados o copiados

2.1. Sincronización de procesos mediante exit() y wait()

La llamada al sistema **exit()** se emplea en aquellos procesos que desean terminar voluntariamente. Tiene un único parámetro entero, denominado estado de salida del proceso. Por convención un estado de salida de 0 indica éxito, y uno mayor que 0 indica fallo. Por otra parte la llamada al sistema **wait()**, se emplea en un proceso padre para esperar por la finalización de uno de sus hijos. La función **wait()** devuelve dos valores:

- El ID del proceso hijo cuya finalización ha originado que wait despierte.
- El estado de salida de ese hijo.

Se pueden ignorar estos detalles e invocar **wait(0)** para que espere porque un hijo finalice.