Sistemas Operativos II
Práctica 2: Shell distribuída.

Escuela Politécnica Superior.
Universidad Autónoma de Madrid.

Noviembre 2006

Calendario

La segunda práctica de esta asignatura se organizará según el siguiente calendario:

Comienzo: Lunes 6 de Noviembre de 2006.

Duración: 6, 13, 20, 27 de Noviembre y 4 de Diciembre de Octubre de 2006.

Entrega: 11 de Diciembre de 2006 **HASTA LAS 13h**.

Examen: 11 de Diciembre de 2006.

Entrega de resultados: Semana del 25 de Diciembre de 2006.

1. Introducción

En esta práctica se va a crear un sistema distribuido sencillo, donde la localización real de un fichero es totalmente irrelevante. La idea general de la práctica es acercarse lo más posible al concepto de transparencia de localización.

El *sistema distribuido* (a partir de ahora llamado *cluster*) consiste en varias copias de un servidor (*dserver*), que serán instanciados, a la vez, en la misma o distintas máquinas. Estos servidores aceptarán peticiones similares a las implementadas en la práctica 1, pero además se comunicarán entre sí para procesar aquellas peticiones que no puedan resolver localmente. Para colaborar entre sí, los servidores deben saber unos de otros, y por tanto tendrán que "darse de alta" en el *cluster* en el que van a participar.

Se deberá desarrollar una "librería" de cliente (libdclient) que permitirá a las aplicaciones que la usen interaccionar con el *cluster*, y operar con archivos y listados que se encuentren accesibles a cualquier máquina de esta unión. El funcionamiento será similar al llevado a cabo en el desarrollo de la primera práctica de la asignatura. La diferencia residirá en que ahora no hará falta especificar el servidor en el que se encuentra el archivo o el listado solicitado, sino que bastará con indicar la dirección de cualquier servidor del *cluster*.

Para probar tanto la librería cliente como los servidores, se deberá desarrollar algunas aplicaciones sencillas que usen *libdclient* para acceder a archivos del *cluster* (dcat, dls y

dtac). Estas aplicaciones, al igual que las de la práctica 1, se deberán poder ejecutar desde línea de comando. No obstante, obtendrán la dirección del dserver a contactar a través de una variable de entorno, en lugar de extraerlo de sus argumentos.

Finalmente, se deberá implementar una "shell" sencilla (dsh) que haga uso de *libdclient* para conseguir una cierta transparencia de localización en la ejecución de procesos. En esta shell, la primera palabra se interpretará como el nombre de un ejecutable remoto. El resto de los argumentos no se interpretan:

```
dsh $ dcat fichero.txt
```

debería mostrar por salida estándar el contenido de `fichero.txt`, aunque el ejecutable *dcat* esté en una máquina del *cluster* y el archivo `fichero.txt` se encuentre en otra, y ninguno de los dos se encuentre accesible localmente en el *dserver* con el cual se comunica *dsh*. (Esto funcionará porque dcat, por su parte, es capaz de leer `fichero.txt` allá donde se encuentre dentro del *cluster*).

2. El servidor: *dserver*

El servidor de esta práctica se comunica con otros servidores que se encuentren en otras direcciones y/o puertos para satisfacer las peticiones de recursos que no están localmente accesibles. Para esto, debe de mantener algún tipo de lista de vecinos con los que comunicarse, y usar las funciones de la librclient desarrollada en la práctica anterior. Hay varias formas de mantener información sobre los vecinos. La más sencilla (y la menos escalable) es la que se ha elegido: cada servidor debe mantener una lista con las direcciones y puertos de los demás servidores del *cluster*. Cuando se arranca un servidor, se debe registrar (mediante una petición d_register) en el *cluster* especificado. Cuando se cierra, deberá darse de baja. Para ello, el ejecutable dserver debe tener los siguientes argumentos de entrada:

dserver «puerto» [«ip y puerto de otro dserver»]

La «ip y puerto de otro dserver» es opcional, y debe tener formato ip:puerto. Si no se especifica, se trata del primer servidor de un *cluster*, y no hace falta ni registrarse ni desregistrarse.

El servidor es una versión ampliada del servidor de la práctica 1. Hace todo lo que puede hacer rserver, pero además se añaden nuevas peticiones:

Llamada y argumentos	**Respuesta**	**Efecto**
r_exec nombre_ejecutable argumentos\n	errno pid_hijo (y stdout: ver abajo)	lanza el ejecutable en la máquina destino; una vez devuelto el valor de retorno, el socket usado se convierte en la entrada y salida del ejecutable.
d_find nombre_fichero\n	errno ip:puerto\n	devuelve la ip y el puerto de un dserver del *cluster* donde existe el fichero solicitado.
d_register ip:puerto\n	errno n ip1:puerto1 ... ipN:puertoN\n	notifica a todas las máquinas del *cluster* que se de que hay un nuevo miembro: el servidor con la ip y puerto suministrados. Como resultado recibe el número de servidores registrados (sin incluir al nuevo) y sus IPs y puertos.
d_unregister ip:puerto\n	errno \n	elimina del *cluster* el servidor con esa ip y puerto. Lo contrario a d_register.
d_ls nombre\n	errno\n	similar a r_ls, pero realiza r_ls's sucesivos sobre todos los servidores que componen el *cluster*

Todas estas peticiones requieren como condición que no haya ningún archivo abierto, y todas ellas cierran la conexión una vez finalizadas (aunque r_exec sólo finaliza cuando el comando ejecutado finaliza). A continuación se describen con más detalle estas operaciones:

2.1. r_exec

Se proporciona una versión ya implementada en `eserver.c` (basta con copiar las funciones relevantes al propio dserver). Para hacer pruebas con `eserver.c`, se debe compilar igual que `s_echo.c` en la práctica anterior, y se puede usar tanto 'telnet' como 'nc' como cliente de pruebas. Es preferible usar 'nc', porque no inserta caracteres '$^\circ$' antes de cada '\n' introducido por stdin. Por ejemplo, suponiendo que *eserver* está funcionando en

localhost:8000, se puede probar como sigue:

```
    $ nc localhost 8000
r_exec /bin/sh
0 11358
echo $DSERVER
127.0.0.1:8000
exit
```

Al conectarse a este servidor y enviar una petición r_exec correcta, se realiza un 'fork()' y un 'execve()' para lanzar el ejecutable solicitado. Si esto funciona, se devuelve un 'errno' de 0 y el PID del hijo obtenido. La salida estándar del ejecutable queda conectada al 'socket' desde el que se realizó la petición, y la entrada estándar del proceso pasa a ser este mismo 'socket'. Las peticiones r_exec permiten interaccionar con un ejecutable remoto de forma muy parecida a como se interacciona con uno local.

Para simplificar la sintaxis de esta petición, se considera que cada palabra a continuación del nombre del ejecutable es un nuevo argumento. Y no se envía ninguna variable de entorno. Como las aplicaciones dcat, dtac, dls y dsh requieren que la variable de entorno DSERVER esté inicializada, ésta se inicializa siempre dentro de la función, con la dirección y el puerto del servidor que recibe la petición r_server.

2.2. d_find

Para buscar un archivo cualquiera, se realizará un *r_open* en cada servidor de la lista. En cuanto uno responda que la apertura ha sido un éxito, se realizará el correspondiente *r_close*, y se devolverá la dirección y el puerto de ése servidor como resultado de la consulta. Da igual que el archivo esté en varios servidores: se devolverá la dirección del primero para el que se encuentre. Si no se encuentra en ninguno, se devolverá como .^{er}rno.^{el} mismo código de error que devuelve `fopen()` en caso de fichero no encontrado.

2.3. d_register y d_unregister

Estas funciones alteran la tabla de "socios del *cluster*"que se mantiene en cada servidor. Debe bastar con solicitar un d_register o un d_unregister una única vez en un único servidor del *cluster* (a uno cualquiera) para que se actualicen las tablas de todos los demás servidores. Esto se puede conseguir reenviando las peticiones correspondientes a aquellos servidores que puede que todavía no la hayan recibido, pero habrá que evitar situaciones en las que este proceso no acabe nunca. En caso de error (intentar borrar algo que no existe, o intentar insertar algo que ya está) deberán devolver EINVAL como número de error.

Para obtener la dirección IP y el puerto de "uno mismo" se puede usar la `consigueDireccionLocal()` suministrada en eserver.c.

2.4. d_ls

Similar a r_ls, en el sentido de cómo se usa: una vez solicitado un d_ls a un servidor a través de un socket, sucesivas peticiones r_read sobre ése socket devolverán fragmentos de los correspondientes 'lss' de cada servidor. En cuanto a implementación, se realizarán

un r_ls() con el mismo argumento sobre los servidores, en orden, realizándose un nuevo r_ls() cada vez que se agota lo que se puede leer del anterior. La implementación de `ls` está muy simplificada en la práctica 1 (habría sido mucho más elegante implementar un r_opendir y un r_readdir); en esta práctica sucede lo mismo - pero sigue siendo útil para hacer diagnóstico.

3. El cliente: *libdclient*

Es muy similar a la librería de la práctica 1. De hecho, la mayor parte de las peticiones se deben implementar mediante llamadas a las correspondientes funciones de *librclient* (*libclient* de la práctica anterior).

La única funcionalidad que es algo distinta al resto es d_register: debe devolver -1 para error, o el número de entradas escrito en direcciones_obtenidas (cada una de ellas de la forma "«ip»:«puerto»\0", en formato texto). Si se devuelve ¿0 (sin errores), entonces direcciones_obtenidas debe contener las direcciones de todos los componentes del *cluster*, incluyendo la ip_y_puerto_a_registrar. Se debe reservar espacio para direcciones_obtenidas antes de llamar a la función, sabiendo que una única dirección no puede ocupar más de 32 bytes y no va a haber más de 128 direcciones. Estas constantes se deberán definir en el .h como MAX_LEN_DIR y MAX_DIRS_CLUSTER, respectivamente.

Funciones de manejo de archivos (ver páginas 'man' de las versiones sin 'd_'):

- `int d_open(char *ip_y_puerto, char *nombre, char *modo);` : ip_y_puerto de cualquier 'dserver'
- `int d_close(int fd);`
- `int d_read(int fd, char *buffer, int number);`
- `int d_seek(int fd, int offset, int whence);`
- `int d_write(int fd, char *buffer, int number);`
- `int d_ls(char *ip_y_puerto, char *directorio);` : `ls` sobre todos los miembros del *cluster*

Nuevas funciones:

- `int d_find(char *nombre_fichero, char *ip_y_puerto, char *resultado);` : 'resultado' debe estar ya reservado
- `int d_exec(char *ip_y_puerto, char *nombre, char* argv[]);` : similar a execv().
- `int d_unregister(char *ip_y_puerto, char *ip_y_puerto_a_eliminar);`
- `int d_register(char *ip_y_puerto, char *ip_y_puerto_a_registrar, char **direcciones_obtenidas);`

Se usará la misma variable global de indicacion de error de *librclient* y no se declarará en el .c

3.1. *d_exec* en el cliente

Una llamada a `d_exec()` debe ser similar a una llamada a `execv()`; por tanto, nada más hacer una llamada a esta función, el cliente deberá quedar bloqueado y la entrada y salida estándares deben pasar a ser las del proceso que se está ejecutando en el servidor. Esto requiere que el proceso actual se bloquee por completo, y:

- todo lo que se reciba por stdin a partir de la llamada a esta función se envíe por el 'socket' correspondiente.
- todo lo que se pueda leer por el socket se muestre inmediatamente por stdout.

Para implementar esto funcionalidad hará falta entrar en un bucle que use `select()` o una combinación de `pollIn()` y `pollOut()` para determinar cuándo hacer lecturas o escrituras al socket. Además, cuando el proceso remoto acabe (determinado porque las lecturas del socket devuelven 0 bytes), se deberá cerrar el extremo local del socket. No se vuelve de un `execve()`, sencillamente se sale del proceso mediante `exit()`.

4. Ejecutables

Los ejecutables `dcat`, `dtac` y `dls` son casi idénticos a los de la práctica anterior. Sólo cambian sus argumentos de entrada y se substituyen todas las funciones llamadas por sus variantes con 'd'. Antes aceptaban argumentos de la forma ip puerto nombre_archivo. Ahora deberán tener sólo el nombre_archivo: la ip y el puerto se deben obtener de la variable de entorno "DSERVER", usando la función `getenv()`.

4.1. La shell distribuida: dsh

Se trata sólo de una demostración, no hay que implementar unaIntegrar todo, y desarrollar más pruebas para verificar que todo funciona como se espera. 'shell' de verdad con todos los comandos y redirecciones y bucles de 'bash'. Esta 'shell':

- Obtiene de la variable de entorno 'DSERVER' la ip y el puerto del servidor con el que va a contactar.
- Interpreta comandos de la forma ¡ejecutable¿¡argumentos¿
- Para cada comando, realiza el `d_exec()` correspondiente y se bloquea hasta que se acaba. Esto se puede realizar mediante un `fork()`, un `d_exec()` en el hijo, y un `waitpid()` bloqueante esperando a que el hijo acabe. Cuando se llame a `d_exec()`, no hace falta pasar ninguna variable de entorno.

Esta shell se podrá ejecutar en modo interactivo o leyendo los comandos de sus argumentos. Para ejecutar dsh en modo interactivo, se llamará sin argumentos. En este caso, pedirá argumentos por entrada hasta que reciba como comando la línea 'exit'. Y en ése momento finalizará. Si se llama con argumentos, el primer argumento será considerado el primer comando, y así sucesivamente. Cuando se quede sin argumentos deberá acabar.

5. Ejercicios optativos

Los ejercicios optativos no serán tenidos en cuenta si los ejercicios obligatorios no funcionan correctamente. En el caso en que haya que modificar el servidor, el cliente o el protocolo de comunicación, asegurarse de que la funcionalidad básica sigue funcionando correctamente.

5.1. transparencia de replicación (+15 %)

En este momento, si hay varias copias de un archivo con el mismo nombre en varios servidores del *cluster*, se eligirá sólo uno de ellos para operar con él. Por tanto, es posible abrir una copia, alterarla, escribirla, y que la vez siguiente uno se encuentre la copia vieja. Propón, sin implementar, pero proporcionando todos los detalles relevantes en la memoria, una serie de cambios de diseño que harían que la práctica cumpliese semántica de sesión a nivel de todo el *cluster*. Los cambios deben limitarse al `dserver` y a `libdclient.c` (sin cambiar para nada `libdclient.h`). No se permitirá añadir nuevos servidores ni servidores centralizados, ni limitarse a borrar todas las copias que se encuentren.

5.2. Pipes y redirecciones (+15 %)

Ampliar los tipos de comandos que puede aceptar `dsh`, de forma que sea posible escribir sentencias de la forma:

dsh$ dcat f.txt >> g.txt
dsh$ dcat f.txt | dtac g.txt
dsh$ dtac g.txt << f.txt

Donde todos estos comandos serían equivalentes, y su resultado final sería copiar el fichero `f.txt` en el fichero `g.txt`, estén donde estén los ejecutables `dcat` y `dtac` y los ficheros `f.txt` y `g.txt`. No tiene por qué existir `g.txt`. Es decir, se ha de permitir que los comandos sean de la forma «comando1» (formato anterior) ó «comando1» «operador» «comando2» (nuevo formato, donde «operador» es <<, |, ó >>). Si el operador es >> ó <<, el «comando2» deberá ser un nombre de fichero en lugar de un ejecutable.

6. Entrega

Se deberá entregar, comprimido en un fichero .tgz:

- El código fuente referente al servidor:

  dserver.c

  dserver.h

- El código fuente correspondiente al api del cliente:

  libdclient.c

  libdclient.h

- El código correspondiente a cada una de las aplicaciones de nivel superior implementadas:

  dls.c

  dcat.c

  dtac.c

  dsh.c

- Los fuentes:

  sock.c y *sock.h*

  semaphore.c y *semaphore.h*

- Un fichero *makefile* que cumpla los siguiente objetivos:

  all: compila todo.

  server: compila el servidor.

  client: compila el cliente.

  apps: compila las distintas aplicaciones.

  test: lanza el servidor y ejecuta un fichero llamado *test.sh* el cual debe ser modificado para llamar a todos los scripts desarrollados.

- Memoria y fichero *leeme.txt*

- Todos los scripts usados para probar la práctica los cuales serán llamados desde *test.sh*

- Un script para sapt llamado *sesion.txt* en el que se demuestre que el funcionamiento de la práctica.

- Cualquier otro script que se considere oportuno.

7. Recomendaciones de implementación

Se recomienda seguir los siguientes pasos en la implementación:

1. Modificar rserver.c y .h para implementar las nuevas características de dserver. Empezando por la funcionalidad de añadir y eliminar miembros del *cluster*, luego incorporando d_find, y finalmente r_exec y r_ls.

2. Desarrollar un script sapt apropiado para probar el servidor, simulando múltiples servidores vecinos.

3. Escribir la librería cliente, libdclient, y escribir dsh para asegurarse de que `d_exec()` funciona como se espera. (Por ejemplo, un `d_exec` de `/bin/sh` debería proporcionar una línea de comandos remota).

4. Implementar el resto de las aplicaciones, lo cual debería resultar sencillo.

5. Integrar todo, y desarrollar más pruebas para verificar que todo funciona como se espera.