Práctica 3. ONC RPC

Arquitectura y Tecnología de Computadores. Sistemas Distribuídos

1. Ejercicios a realizar

Los ejercicios a realizar están basados en los realizados durante las sesiones 4 y 5 de prácticas.

1.1. Cliente/servidor desarrollados con el mecanismo rpcgen

Escribir un interfaz (incluyendo las definiciones de tipos necesarias) para los servicios siguientes:

Servicio	**Recibe**	**Retorna**	**Hace**
`SumaComplex`	Dos números complejos	Un número complejo	Suma los dos complejos recibidos como parámetro
`Acaba`	Nada	Nada	"Desregistra" el programa y finaliza su ejecución
`Media`	Array variable de reales	Media aritmética o error	Calcula la media aritmética, salvo si el array está vacío

Como número de programa se utilizará un número hexadecimal de ocho cifras, siendo la primera un "2" y las siete restantes las últimas cifras del DNI del alumno. El número de versión y los números de servicio son libres, salvo para el servicio `Acaba` que debe tener el número 1.

Escribir un servidor que implemente estos servicios. El error retornado por la función `Media()` consiste en una cadena de texto con un mensaje del tipo "Error, el array no tiene elementos".

Escribir un cliente que invoque los servicios `SumaComplex()` y `Media()` con datos proporcionados por el usuario a través del teclado. Los complejos serán de la forma $a + b\mathbf{i}$ y en el caso del array de longitud variable se preguntará antes por el número de elementos que contiene. El array que se envíe al servidor no debe tener más elementos de los que el usuario haya especificado.

El cliente debe además recibir por línea de comandos el nombre de la máquina en la que se está ejecutando el servidor.

1.2. Cliente desarrollado con el mecanismo simplificado

Escribir un cliente que invoque el servicio `Acaba()` utilizando el mecanismo de RPC simplificado[1]. Este cliente recibirá por línea de comandos el número de programa a "acabar", que se interpretará en hexadecimal[2]. Por ejemplo, la invocación siguiente acabará con la ejecución del programa de número `0x28767812`.

```
./acabador 28767812
```

Probar este programa para "acabar" el servidor anterior. Comprobar con `rpcinfo` que efectivamente se elimina del localizador y con `ps` que deja de ejecutarse.

[1] Aunque el servidor esté desarrollado con rpcgen, sus servicios se pueden igualmente invocar con el mecanismo de RPC simplificado, siempre que se conozcan los números de programa, versión y función.

[2] `scanf` puede leer números en hexadecimal utilizando el formato `" %x"`

1.3. Autenticación

Modificar el servidor del punto 1.1 para que sólo permita la ejecución del servicio "acabar" al usuario que es el propio alumno, y sólo cuando esta petición provenga de la máquina "orion"[3].

Intenta "acabar" el nuevo servidor con el cliente desarrollado en el punto 1.2 y comprueba que no se puede (debido a que este cliente no aporta autenticación). Desarrolla un nuevo cliente "acabador", esta vez utilizando el mecanismo `rpcgen`, que proporcione las credenciales correctas y comprueba que ahora sí es capaz de "acabar" nuestro programa, cuando lo ejecutas desde `orion`.

2. Aspectos formales

Debes crear una carpeta llamada `p3` y dentro de ella dos carpetas llamadas `e1`, `e2` y `e3`, una para cada uno de los ejercicios anteriores. Cada carpeta debe contener el código fuente (con los nombres de ficheros que detallaremos seguidamente) y alguna forma automatizada de compilación, ya sea en forma de *script* o de `Makefile`. El código fuente debe ser C estándar y debe compilar y funcionar al menos en las dos máquinas de prácticas `sirio` y `orion`.

Los nombres de los ficheros fuente serán los siguientes:

Ejercicio 1 `interfaz.x`, `servicios.c` y `cliente.c`

Ejercicio 2 `acabador.c`

Ejercicio 3 `interfaz.x` (es el mismo que en el ejercicio 1), `servicios2.c` y `acabador2.c`

2.1. Entrega

Cuando el código haya sido probado, se borrarán los ejecutables, dejando sólo el código fuente y el *script* o `Makefile` de compilación, y se empaquetarán todas las carpetas en un único archivo `tar` comprimido. Para esto, debes situarte en la carpeta "padre" de la carpeta `p3` y desde ella teclear:

```
$ tar czvf p3.tar.gz p3
```

El resultado será el fichero `p3.tar.gz`, que seguidamente deberás firmar digitalmente con el comando[4]:

```
$ gpg --sign p3.tar.gz
```

El resultado será un archivo `p3.tar.gz.pgp` que entregarás a través del formulario Web en la página de la asignatura. Si lo deseas, además de firmarlo puedes cifrarlo para tu profesor de prácticas, mediante el comando:

```
$ gpg --sign --encrypt -r jldiaz@uniovi.es -r ariasjr@uniovi.es p3.tar.gz
```

3. Anexo: detalles de funcionamiento y aclaraciones

Se detallan en este anexo algunas aclaraciones sobre diferentes aspectos que suelen causar confusión a la hora de realizar prácticas con rpcgen en las máquinas de prácticas `sirio` y `orion`.

[3] Observa que en diferentes máquinas puedes tener diferentes *uid*.

[4] Deberás efectuar esta operación en la máquina en la que hayas creado tu pareja de claves GPG

3.1. Funcionamiento del servidor en `sirio`

El stub de servidor generado por `rpcgen` en sirio hace que este servidor tenga un comportamiento diferente al que tendría en una máquina Linux. En particular:

- Los servicios se registran bajo protocolos específicos de Sun, además de los estándares TCP y UDP. La función `pmap_unset` desregistra sólo los estándares, por lo que es normal que tras llamar al servicio `acaba` se sigan viendo los otros protocolos registrados en la tabla mostrada por `rpcinfo`. Si se quiere evitar ésto (aunque no es un fallo que se tenga en cuenta), úsese la opción `-n tcp -n udp` y de ese modo el servidor generado por `rpcgen` sólo registrará los servicios bajo TCP y UDP.
- El servidor, tras arrancar, pasa a tarea de segundo plano. Esto tiene dos efectos secundarios: si usabas `printf` en el servidor para depurar, no saldrán por pantalla. Además, el proceso no aparece al ejecutar `ps`. Para ver si el proceso está en ejecución, debes usar `ps -ef` que te muestra todos los procesos del sistema, y buscar los tuyos con `grep`, por ejemplo:

```
ps -ef|grep uo1263721
```

3.2. El error del cliente acabador

Cuando el cliente `acabador` se ejecuta, es normal que termine con un error, ya que el servicio `acaba`, finaliza el proceso servidor sin darle oportunidad de enviar al cliente la respuesta apropiada. El cliente por tanto, tras un tiempo de espera, muestra un error.

Sin embargo, el error no ocurre cuando el servidor se está ejecutando en una máquina Linux. Parece que el servidor, o el operativo, envía algún tipo de respuesta incluso después de finalizar, pero desconozco el mecanismo por el que lo logra.

3.3. La entrega

La sintaxis del servicio (su prototipo) es diferente bajo SunOS que bajo Linux. Aparentemente esto implica que el fichero `servicios.c` debe ser diferente en Sirio y en Orion. Lo cierto es que no tiene por qué ser así. Puede hacerse un solo fichero que compile correctamente en ambas arquitecturas, si se hace uso de las directivas de compilación condicional (`#ifdef/#endif`), sabiendo además que el compilador define automáticamente la constante `__sun` cuando se compila bajo SunOS (sirio), y la constante `linux` cuando se compila en Linux (orion).

No obstante, si no sabes cómo usar estas directivas, admitiremos también la entrega de dos ficheros diferentes, llamados `servicios-sirio.c` y `servicios-orion.c`. Lo que sí debes proporcionar es un `Makefile` adecuado a cada máquina que compile la versión apropiada. O también, si lo prefieres, un Makefile único que detecte en qué máquina está (puedes hacer uso de la instrucción `ARQUITECTURA=$(exec uname)` dentro del `Makefile`, para definir la variable `ARQUITECTURA`, que tomaría el valor `SunOS` o `Linux`).