Sistemas Operativos Distribuidos
Ejercicios de Examen

Jose Luis Díaz

Extraídos del curso 2004-2005

Estos son ejercicios procedentes de exámenes de la asignatura. Los problemas aquí mostrados son del tipo de los que requieren operaciones y respuesta numérica, o los que consisten en completar listados. En los exámenes también aparecen preguntas de teoría, de desarrollo un poco más largo, similares a las preguntas 15 y 14 que se incluyen a modo de ejemplo en esta colección.

Problema 1.

El siguiente fragmento de código implementa un servidor mediante la programación directa de los sockets, usando el protocolo TCP. No se muestra el código de una función, `RealizarServicio()`, que sería la que recibiría los datos del cliente y le enviará la respuesta. Esta función requiere como parámetro el socket a través del cual se comunicará con el cliente. Parte del código está tapado y se preguntará más adelante sobre él. No se incluye el código necesario para chequear errores e imprimirlos.

```
1  // #includes omitidos
2  #define PUERTO 12345
3  int main() {
4   /* Declaraciones */
5    int sConex, sDat;
6    char buff[100];
7    struct sockaddr_in Servidor, Cliente;
8    int ldir=sizeof(struct sockaddr_in);
9
10   /* Inicializar sconex */
11   sConex=
12
13   /* Asignarle puerto y dirección INETADDR_ANY) */
14        .sin_family=AF_INET;
15        .sin_port=
16        .sin_addr.s_addr=
17   bind(sConex,                              );
18
19   /* Poner en modo escucha */
20   listen(sConex, SOMAXCONN);
21
22   /* Bucle infinito para atender clientes */
23   while(1) {
24     =
25    RealizarServicio(sDat);
26    close(    );
27   }
28  }
```

❑ 1.1 Completa la línea con la inicialización de sConex (línea 11)

Respuesta:

Se trata de la creación del socket con los parámetros habituales. Ya que se trata de un socket para el protocolo TCP, debe especificarse la constante `SOCK_STREAM`.

Solución:

```
sConex=socket(PF_INET, SOCK_STREAM, 0);
```

❑ 1.2 Completa el fragmento de código entre las líneas 14 y 17.

Respuesta:

Se trata de la inicialización de la estructura que contiene la IP y puerto asignados al socket. Esta estructura es la variable `Servidor` (de tipo `struct sockaddr_in`). Es importante no olvidar las macros `htons` y `htonl` para que el orden de los bytes sea el correcto. La función `bind()` asigna esta dirección al socket, para lo cual se le pasa como segundo parámetro la dirección de la estructura y como tercer parámetro su tamaño.

Solución:

```
Servidor.sin_family=AF_INET;
Servidor.sin_port=htons(PUERTO);
Servidor.sin_addr.s_addr=htonl(INADDR_ANY);
bind(sConex,&Servidor,sizeof(Servidor));
```

❑ **1.3** Completa el código del `while(1)`

Respuesta:

El bucle debe esperar a que llegue un cliente y seguidamente darle servicio, tras lo cual podrá cerrar el socket de datos con el que se comunicaba con el cliente. Por tanto la primera línea que falta debe ser una llamada a la función `accept()`, para esperar por clientes. Esta función retorna un socket de datos recién creado para comunicarse con ese cliente, que deberemos asignar a la variable `sDat`, puesto que ese es el parámetro que seguidamente se le pasa a la función `RealizarServicio()`. Ese mismo socket (`sDat`) será el que hay que cerrar con `close()` antes de repetir el bucle para esperar por un nuevo cliente.

Solución:

```
sDat=accept(sConex, &Cliente, &ldir);
RealizarServicio(sDat);
close(sDat);
```

❑ **1.4** Si quisiéramos portar el programa anterior a Windows, habría que añadir unas llamadas para inicializar la librería WinSock y además modificar algunas líneas del código anterior. ¿Cuáles habría que modificar? Escríbelas primero tal como figuran en el listado anterior y al lado cómo quedarían modificadas.

Solución:

Original:	`int sConex, sDat;`
Nueva:	`SOCKET sConex, sDat;`
Original:	`close(sDat);`
Nueva:	`closesocket(sDat);`

Problema 2.

Un programa ha creado 5 sockets y los ha almacenado en un array. Cada socket ha sido asignado a un puerto diferente, y se le ha puesto en modo escucha. Se desea que seguidamente el servidor quede a la espera de clientes en cualquiera de los sockets, y según el socket que reciba la conexión se ejecute una función u otra. Para ello, la función `EjecutarServicio()` recibe un entero, entre 0 y 4, que indica qué servicio se debe ejecutar. El código de esta función se muestra en un cuadro aparte.

```
1  /* Variable global */
2  int sockets[5];
3  ...
4  main() {
5   /* Variables locales de main */
6   fd_set cjto;
7   int max;
8
9  /* No se muestra la inicialización de los sockets */
10 while (1) {
11  FD_ZERO(&cjto);
12  max = 0;
13  for (i=0; i<5; i++) {
14     [                          ]
15     [                                    ]
16  }
17  /* Esperar conexiones */
18  select(max+1, &cjto, NULL, NULL, NULL);
19  /* Elegir función, según el socket que
20    haya recibido la conexión */
21  for (i=0; i<5; i++) {
22    if ([       ]([         ], [    ]))
23       EjecutarServicio(i);
24   } // for
25  } // while
26 } // main
```

```
1  // Funcion EjecutarServicio()
2  void EjecutarServicio(int n) {
3    int sDat;
4    sDat=accept(sockets[n], NULL, 0);
5    switch(n) {
6     case 1: Servicio1(sDat); break;
7     case 2: Servicio2(sDat); break;
8     ...
9     default: fprintf(stderr, "Error\n");
10   }
11   close(sDat);
12 }
```

❑ **2.1** Completa el primer bucle for (línea 14 y siguiente)

Respuesta:

Antes de invocar `select()`, es necesario inicializar una variable que indique el valor numérico del socket con un descriptor más alto (más uno), y un mapa de bits indicando qué sockets queremos "observar". El bucle `for` inicial, por tanto, tiene como cometido la inicialización de ambas variables.

El mapa de bits debe ser una variable de tipo `fd_set`, y por tanto será la variable `cjto`. Esta variable se inicializa mediante las macros `FD_ZERO` (para ponerla completamente a cero) y `FD_SET` para poner a 1 los bits correspondientes a los sockets que se quieren observar, que son justamente los almacenados en el array `sockets[]`. Por otro lado, si observamos la llamada a `select()` en la línea 18, vemos que el primer parámetro que se le pasa es `max+1`, por lo que la variable `max` debe ser inicializada simplemente con el mayor de los elementos de `sockets[]`.

A partir de estas ideas no es difícil completar el código pedido.

Solución:

```
FD_SET(sockets[i], &cjto);
if (max<sockets[i]) max=sockets[i];
```

❑ **2.2** Completa el `if` de la línea 22

Respuesta:

Una vez se ha salido del `select()`, la variable `cjto` ha cambiado de valor, y ahora contiene un 1 sólo en el socket en el que se ha detectado actividad (esto es, un intento de conexión). La macro `FD_ISSET` permite averiguar si un socket dado está a 1 o no en el mapa de bits. Comprobando con esta macro todos los sockets almacenados en el array `sockets[]` determinaremos cuál ha tenido la actividad, y llamaremos a la rutina de servicio pasándole el índice del array de ese socket. A partir de estas consideraciones, es directo determinar la línea que falta.

Solución:

```
if (FD_ISSET(sockets[i], &cjto))
```

Problema 3.

¿Cuál o cuáles de las siguientes afirmaciones son ciertas?

A) La función `bind` debe usarse en el cliente para asignar la IP y puerto del servidor antes de conectar.

B) La función `htons` convierte un dato de tipo `short int` del orden de máquina al orden de red.

C) En algunas máquinas la función `htonl` no tiene efecto alguno (devuelve el mismo número que recibe)

D) Si un proceso crea un socket y seguidamente varios hijos que ejecutan un `accept()` sobre ese socket, cuando llegue un cliente todos los hijos le atenderán.

Respuesta:

La afirmación A es falsa, puesto que `bind()` es para asignar una dirección al socket *local*. La dirección del otro extremo se especifica en el `connect()`. La afirmación D también es falsa, puesto que sólo uno de los hijos es desbloqueado. B y C son ciertas.

Solución:

```
B y C
```

Problema 4.

En un fichero XDR se declaran los siguientes tipos:

```
1 struct coord {
2    int x;
3    int y;
4 };
5 typedef coord Puntos<100>;
```

En un programa en C se crea una variable de tipo `Puntos` para contener dos elementos que se inicializan como se muestra en el código siguiente:

```
1 Puntos p;
2 ...
3 p.Puntos_len=2;
4 p.Puntos_val=malloc(2*sizeof(coord));
5 for (i=0; i<2; i++) {
6   p.Puntos_val[i].x=i+1;
7   p.Puntos_val[i].y=i*2;
8 }
```

❑ **4.1** La variable p se vuelca a un fichero, codificado en XDR. ¿Cuáles serán los 12 primeros bytes almacenados en el fichero? Responde en hexadecimal cada byte.

Respuesta:

El tipo `Puntos` es un array de longitud variable y por tanto su codificación comenzará indicando el número de elementos. Este número es dos, debido a la línea 3 que asigna `p.Puntos_len`. Así pues los primeros cuatro bytes de la codificación serán `00 00 00 02`. Seguidamente debe ir cada elemento del array. En este caso cada elemento es una estructura y por tanto se codificarán sus campos en secuencia, primero el campo `x` y después el campo `y`. Analizando el fuente C vemos que el campo `x` va tomando los valores 1, 2, ... mientras que el campo `y` toma los valores 0, 2, ... Por tanto la primera estructura en el array almacena los datos `x=1`, `y=0` y la segunda estructura almacena `x=2`, `y=2`. Cada uno de estos datos es un entero que se codifica con cuatro bytes, lo que nos da finalmente la respuesta.

Solución:

```
                                        02 00 00 00
02 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00
```

❑ **4.2** ¿Cuál es el mínimo número de bytes que puede ocupar una variable de tipo `Puntos` una vez codificada en XDR?

Respuesta:

Ya que `Puntos` es un array variable, su tamaño mínimo ocurrirá para el caso en que contenga cero elementos. En este caso su codificación consistirá tan solo en el contador de elementos, que ocupa cuatro bytes (que serían `00 00 00 00`).

Solución:

```
4
```

Problema 5.

El siguiente fragmento de código muestra parte de un cliente que recibe a través de un socket un dato de tipo `Puntos`, para lo cual usa un filtro XDR. El programa tiene partes ocultas sobre las que se preguntará a continuación

```
1  /* Declaración de variables *
2  int socketDatos;
3  FILE *fsock;
4  XDR op;
5  Puntos p;
6
7  /* La inicialización del socket y conexión con el servidor no se
8  muestra. Suponer que ya está conectado a través de sockDatos */
9
10 /* Inicialización de variables */
11 fsock=                          
12 xdrstdio_create(                );
13 p.Puntos_val=NULL;
14
15 /* Llamada a filtro para leer datos */
16 if (                !=TRUE) {
17   fprintf(stderr, "Error al recibir\n");
18   exit(-1);
19 }
20
21 /* Uso de los datos recibidos, no se muestra */
22
23 /* Liberar mem. reservada por filtro */
24 xdr_    (              );
```

❑ 5.1 Completa los huecos de la línea 11

Respuesta:

Se trata de asociar la acción del filtro con el socket `socketDatos`. Para asociar un socket con un filtro, primero el descriptor de socket debe ser convertido en un descriptor de fichero, lo que se logra llamando a la función `fdopen()`, en este caso para lectura (modo `"r"`). Seguidamente ya se puede usar la función `xdrstdio_create()` para asociar la operación (`op`) con el descriptor de fichero recién abierto (`fsock`) y para la operación de decodificación (`XDR_DECODE`).

Solución:

```
fsock=fdopen(socketDatos, "r");
xdrstdio_create(&op, fsock, XDR_DECODE);
```

❑ 5.2 ¿Qué falta en el hueco de la línea 16?

Respuesta:

Se trata de la llamada al filtro XDR que codificaría la variable de tipo `Puntos`. Este filtro se llama por tanto `xdr_Puntos()` y recibe, como todos los filtros XDR, un primer parámetro indicando la operación (`op`) y un segundo parámetro indicando la variable a flitrar, que en este caso será `p`. Ambos se pasan por referencia.

El `if` en el que esta llamada queda incrustada sirve para comprobar si el flitro ha funcionado o ha generado un error, ya que en este segundo caso retornaría `FALSE`.

Solución:

```
if (xdr_Puntos(&op, &p)!=TRUE)
```

❑ 5.3 Completa el hueco de la línea 24

Respuesta:

Tal como reza el comentario de la línea anterior, se trata de liberar la memoria reservada por el filtro. En efecto, cuando un filtro XDR decodifica datos con una estructura dinámica tal como una lista enlazada, va reservando memoria para cada nodo de la lista a medida que va recibiendo estos datos. El filtro retorna finalmente un puntero al primer elemento de la lista.

Podríamos escribir un bucle que vaya liberando la memoria ocupada por cada uno de los nodos, pero mucho más sencillo resulta llamar a la función `xdr_free()`, cuyo cometido precisamente es liberar la memoria ocupada por cualquier tipo de datos dinámico que haya sido creado por un filtro XDR. A esta función hay que pasarle en primer lugar el nombre del filtro que creó los datos (`xdr_Puntos`) y en segundo lugar la dirección donde comienzan estos datos (`&p`).

Solución:

```
xdr_free(xdr_Puntos, &p);
```

Problema 6.

Considera la siguiente declaración XDR:

```
1 struct Grupo {
2   int numeros<10>;
3   string *texto<13>;
4 } ;
```

¿Cuál sería el máximo tamaño de un dato de este tipo, una vez codificado en XDR? ¿Y su tamaño mínimo?

Respuesta:

El tamaño máximo ocurre cuando ambos datos, que son de longitud variable, alcanzan su longitud máxima. En este caso el campo `numeros` contendrá 10 enteros, y se codificará con 44 bytes (4 para el contador y 40 para los datos). El campo `texto`, que es opcional (obsérvese el `*`) debe estar presente, por lo que ocupará 4 bytes para el indicador de que *sí* hay dato, más lo que ocupe el dato en sí, que en este caso, al ser una cadena de 13 letras serán 20 bytes (4 para el contador, 13 para las letras, y 3 de relleno para alcanzar el siguiente múltiplo de 4). De modo que la estructura completa ocupará $44 + 24$ bytes.

El tamaño mínimo ocurre cuando el array variable `numeros` tiene cero elementos y el campo opcional `texto` no está presente. En este caso se requieren 4 bytes para el primer campo (para almacenar el contador indicando que hay 0 elementos) y 4 bytes para el segundo (para almacenar el indicador 00 00 00 00 de que *no* hay dato opcional).

Solución:

```
MAXIMO = 68      MINIMO = 8
```

Problema 7.

A continuación se muestra parte del contenido de un fichero de cabecera llamado `datos.h` generado por `rpcgen`.

```
 1 struct play {
 2    int n;
 3    struct {
 4       unsigned int data_len;
 5       char * data_val;
 6    } data;
 7    struct {
 8       unsigned int val_len;
 9       int *val_val;
10    } val;
11    char *ranking;
12    miunion foo;
13 }
```

¿Cúal ha sido el código del lenguaje XDR que ha dado lugar a esa salida?. Nota: la palabra `opaque` aparece al menos una vez en el fichero.

Respuesta:

En el fichero `datos.h` vemos la declaración de un tipo `struct`, por tanto el original XDR pudo ser a su vez un `struc`, o también un array variable o una unión, dado que estos tipos también generan un `struct`. Sin embargo, si estuvieramos ante un array de longitud variable el primer campo de la estructura tendría un nombre terminado en `_len`, lo que no es el caso, y si estuvieramos en el caso de una unión discriminada el primer campo de la estructura sería un discriminante (lo que en principio podría ser el `int n;`), y el segundo campo tendría que ser una unión, con nombre terminado en `_u`, lo que tampoco es el caso.

Por tanto el tipo de datos original del fichero XDR tiene que ser una estructura, de nombre `play` al igual que en `datos.h`. Debemos averiguar seguidamente el tipo y nombre de cada campo de la estructura XDR, basándonos en los tipos y nombres de los campos que encontramos en el `.h`.

- El primer campo es `int n;` lo que significa que el primer campo de la estructura XDR también era un entero llamado `n`.
- El segundo campo es un `struct`, pero por la estructura vemos que se trata de la conversión a C de un array de longitud variable. El tipo de dato a que apunta el campo `data_val` es `char`, lo que puede significar que en el original XDR teníamos un array variable de `char`, o de opacos. De momento esto es una incógnita.
- El tercer campo es similar al anterior, pero ahora está claro que el array original era de enteros.
- El cuarto campo es un `char *` y esto sólo puede originarse si el campo original XDR era de tipo `string`.
- Finalmente, el último campo es de tipo `miunion` que se supone que es un tipo definido previamente, y del cual no necesitamos saber su definición. La declaración XDR será exactamente la misma.

A partir de las consideraciones anteriores, la única duda es si el segundo campo es un array de longitud variable de caracteres o de opacos. Pero ya que el enunciado dice que la palabra `opaque` debe aparecer, la duda queda despejada.

Solución:

```
struct play {
   int n;
   opaque data<>;
   int val<>;
   string ranking<>;
   miunion foo;
}
```

Problema 8.

Se tienen tres máquinas, que llamaremos *A*, *B* y *C*, cuyos relojes no van sincronizados. La tabla siguiente muestra una serie de eventos que ocurren en estas tres máquinas, y para cada evento se muestra el valor que el contador de interrupciones tiene en la máquina en que el evento ha ocurrido.

Maquina	Evento	Contador	Descripción
A	A_1	2	Envía mensaje a B_1
	A_2	7	Usuario toca tecla
	A_3	16	Recibe mensaje de B_3
	A_4	25	Envía mensaje a C_3
B	B_1	3	Recibe mensaje de A_1
	B_2	7	Recibe mensaje de C_1
	B_3	9	Envía mensaje a A_3
C	C_1	15	Envía mensaje a B_2
	C_2	19	Usuario toca tecla
	C_3	24	Recibe mensaje de A_4

Se pide aplicar el algoritmo de Lamport para sincronizar los relojes lógicos de estas tres máquinas y seguidamente responder a las siguientes cuestiones.

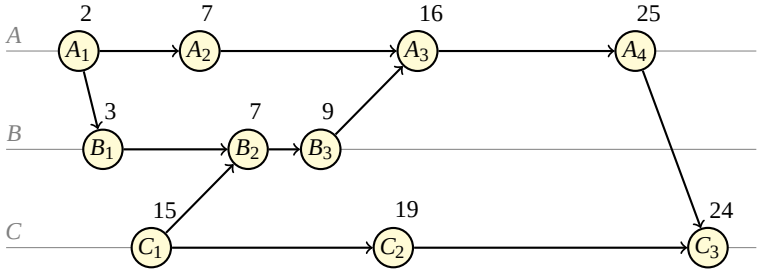
❑ **8.1** ¿Qué valor tendrá el contador de la máquina C en el evento C_3, una vez aplicado el algoritmo?

Respuesta:

A partir de los datos de la tabla del enunciado, es posible construir un gráfico en el que queden reflejadas las relaciones de precedencia entre los eventos. En este gráfico dibujaremos una flecha entre dos eventos si podemos garantizar que el evento al inicio de la flecha ocurrió antes que el evento al final de la flecha. Esto sólo lo podremos garantizar en dos casos:

1. Que ambos eventos ocurran en la misma máquina, y el contador de esa máquina sea mayor para el segundo evento. Así tendremos que $A_1 \to A_2 \to A_3 \to A_4$ y análogamente $B_1 \to B_2 \to B_3$ y $C_1 \to C_2 \to C_3$.
2. Que el primer evento sea el envío de un mensaje y el segundo sea su recepción. En el caso de este enunciado tendremos $A_1 \to B_1, A_4 \to C_3, B_3 \to A_3$ y $C_1 \to B_2$.

Si dibujamos este grafo y ponemos en cada evento el valor del contador en la máquina correspondiente, tendremos la figura siguiente (aún no hemos aplicado el algoritmo de Lamport):



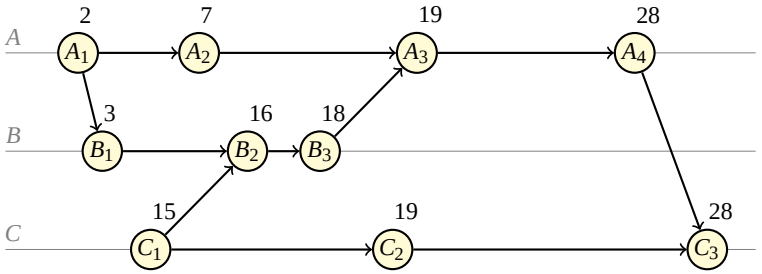
Seguidamente recorremos el grafo de izquierda a derecha, buscando contradicciones en los eventos que implican un envío de mensaje. El primero de estos eventos es $A_1 \to B_1$ en el que no hay contradicción, ya que el contador de B_1 es mayor que el de A_1.

El siguiente evento a examinar es $C_1 \to B_2$, en el que detectamos contradicción. Para evitarla, es necesario adelantar el reloj de B de modo que en B_2 valga 16 (uno más que el de C_1). Esto implica sumarle 9 al reloj de B, ya que antes valía 7. Téngase en cuenta que este incremento de 9 unidades afecta a todos los eventos futuros de B. En particular, el evento B_3 tendrá ahora un contador de 18, en lugar de 9.

El siguiente evento a examinar es $B_3 \to A_3$. Si tenemos en cuenta que, tras el ajuste anterior, el contador de B_3 es 18, detectamos una nueva inconsistencia, ya que el de A_3 sería menor. Por tanto hay que adelantar el reloj de A de modo que en A_3 tome el valor 19 (el de B_3 más uno). Esto implica adelantar 3 unidades el reloj de A, lo que de nuevo afecta a los eventos futuros (A_4 tendría ahora un contador de 28).

Finalmente, el último evento de mensaje a analizar es el $A_4 \to C_3$. Teniendo en cuenta el ajuste anterior, el contador de A_4 vale 28, por lo que habrá que adelantar el reloj de C de modo que en C_3 valga 29 (uno más que el de A_4), lo que nos da la respuesta final.

El siguiente gráfico resume cómo quedan los contadores de las máquinas tras aplicar el algoritmo de Lamport.



Solución:

29

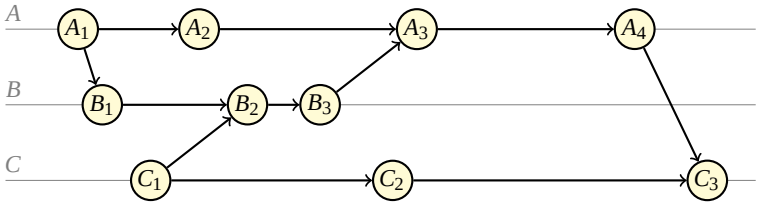
❑ **8.2** Indica cuáles de las siguientes relaciones son ciertas

A) $A_2 \to B_3$
B) $B_1 || C_2$
C) $C_1 \to A_3$
D) $A_3 || C_3$
E) $B_2 \to C_3$

Respuesta:

Para responder a esta pregunta ni siquiera son necesarios los valores de los contadores, ya que el hecho de que un $X \to Y$ no depende de los valores de los contadores, sino tan sólo de si podemos encontrar una cadena lógica de implicaciones que demuestren que X ocurrió antes que Y.

En particular, si simplemente borramos los números de las figuras anteriores y nos quedamos con el grafo, el hecho de que dos eventos X e Y estén conectados por un camino de flechas que llevan de X a Y es una demostración de que $X \to Y$. Si no podemos encontrar tal camino, y tampoco uno que lleve de Y a X, entonces no puede demostrarse qué evento ocurrió primero y en tal caso diremos que son concurrentes, lo que se representa por $X || Y$.



Por ejemplo, observando la figura es fácil ver que hay un camino desde C_1 hasta A_3, que es el camino $C_1 \to B_2 \to B_3 \to A_3$, lo cual demuestra que $C_1 \to A_3$ y por tanto la opción C) es correcta. También es fácil comprobar que no hay camino desde B_1 hacia C_2, y tampoco a la inversa, por lo que $B_1 || C_2$, y por tanto la opción B) es cierta. De forma análoga se encuentra la veracidad o falsedad de las demás opciones.

Solución:

b‘c‘d

Problema 9.

Un servidor para el servicio `time`, tiene un reloj que presenta un ratio máximo de deriva de 0.001. También dispone de un receptor de WWV. Si se ha sincronizado con UTC (usando el receptor) a las 15:00:00 ¿a qué hora debería sincronizarse de nuevo si no quiere que su error sea superior a 1 segundo?

Respuesta:

El ratio de deriva es la velocidad a la que su reloj se aparta de la hora correcta. El valor 0.001 indica que en un segundo se habrá atrasado o adelantado como máximo una milésima de segundo. Por tanto, tras 1000 segundos el retraso o adelanto acumulado puede llegar a ser de 1 segundo y, aunque también podría ser menor, será necesario resincronizar si queremos estar seguros de que nunca se sobrepasa este error.

Así pues, este reloj debería resincronizarse cada 1000 segundos como máximo. Si se ha sincronizado por última vez a las 15:00:00 tendrá que hacerlo de nuevo a las 15:16:40.

Obsérvese que este no es el mismo caso que el que se explica en las transparencias de teoría. En el caso de este problema, tenemos un reloj con deriva que se sincroniza frente a otro que es exacto, mientras que en el caso explicado en la teoría se tienen *dos relojes con deriva* y lo que se pretende es que estos relojes nunca estén separados más de una cantidad dada entre sí. En el caso de las transparencias, la frecuencia de sincronización debería ser el doble, ya que en el peor de los casos uno de los relojes podría estar adelantando mientras que el otro está atrasando. Tras 500 segundos podría darse el caso en que uno hubiera adelantado 0.5s mientras que el otro hubiera atrasado 0.5s, por lo que su distancia relativa podría llegar a ser de 1s, y sería el momento de resincronizarlos con un reloj exacto.

Solución:

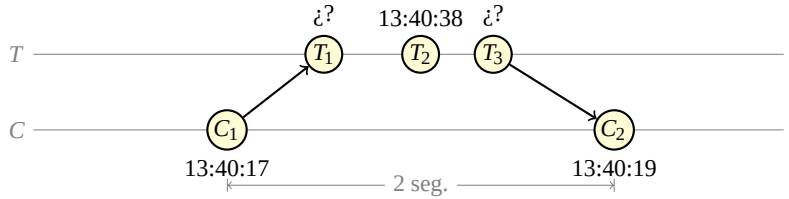
1000 segundos después, esto es, a las 15:16:40

Problema 10.

Un computador quiere sincronizarse con otro que mantiene la hora exacta. Para ello se conecta por TCP con el servicio "`time`", y 2 segundos más tarde recibe la respuesta. Su reloj local marcaba las 13:40:17 cuando hizo la consulta, y la respuesta del servidor contiene la hora 13:40:38. ¿Cuántos segundos debe avanzar su hora al recibir la respuesta? Razónalo.

Respuesta:

Llamemos C al computador que quiere sincronizarse y T al que tiene la hora exacta. Los eventos involucrados en el problema se representan en la siguiente figura y se explican seguidamente:



C_1 El ordenador C envía un mensaje a T, intentando la conexión con el servicio "`time`". En ese instante su reloj marcaba las 13:40:17

T_1 El ordenador T recibe el mensaje de C, aceptando la conexión. No sabemos la hora ni en C ni en T de este evento.

T_2 El ordenador T averigua qué hora es, y obtiene las 13:40:38 (sabemos este dato, puesto que es el que le envía al cliente). No sabemos en cambio que hora es en C mientras tanto.

T_3 El ordenador T envía el mensaje con la respuesta a C. Desconocemos la hora tanto en T como en C de este evento.

C_2 El ordeandor C recibe el mensaje con la respuesta. En ese momento el reloj de C marca las 13:40:19 (ya que el enunciado dice que recibe la respuesta dos segundos después). No sabemos lo que marca el reloj de T en ese instante.

La hora que C obtiene en el mensaje no es la hora exacta para ese instante. Era la hora exacta en el instante T_2, pero desde entonces ha transcurrido una cantidad de tiempo desconocida.

Lo que dice el algoritmo de Crhistian es que, en ausencia de más información, es una buena aproximación suponer que el instante T_2 ocurre justo en el punto medio entre C_1 y C_2. Es decir, la distancia entre T_2 y C_2 sería en este caso de 1 segundo.

Podemos solucionar entonces el problema con dos enfoques diferentes:

- Podemos razonar qué hora era en C en el instante en que en T eran las 13:40:38, y así deducir qué retraso tiene C. En este caso serían las 13:40:18 en C, lo que significa que lleva un retraso de 20 segundos. Esta sería la cantidad a sumar a su reloj para ponerse en hora, y por tanto la respuesta del problema.
- También podemos razonar que, ya que la distancia entre T_2 y C_2 es de 1 segundo en este caso, la máquina C simplemente deberá actualizar su reloj para que en C_2 sea igual a $T_2 + 1$seg. (en general, habrá de ser $T_2 + \frac{1}{2}(C_2 - C_1)$).

Solución:

El algoritmo de Chrisitan supone que cuando el servidor envió la respuesta eran las 13:40:18 local (la mitad entre la hora de petición y la de llegada). Pero eran las 13:40:38 en el servidor, por tanto hay que adelantar 20 segundos

Problema 11.

El siguiente listado muestra la declaración de un interfaz ONC RPC, en lenguaje XDR.

```
1  typedef float Datos<>;
2  union Respuesta switch(int tipo) {
3    case 1: float media;
4    default: string error<>;
5  };
6  program SERVIDOR {
7    version BETA {
8      Respuesta promediar(Datos) = 1 ;
9    } = 2;
10 } = 0xA0001234;
```

Seguidamente se proporciona el código de un cliente que invoca al procedimiento remoto `promediar()` y muestra el resultado. El código tiene ocultas algunas partes por las que se preguntará a continuación.

```
1  Datos dat;
2  Respuesta *r;
3  CLIENT *conexion;
4
5  /* Inicializacion de "conexion" omitida */
6  /* Inicialización de "dat" omitida */
7  /* Ahora viene la invocación remota*/
8   [ ] = [          ]( [   ], conexion );
9
10 /* Seguidamente se muestra la respuesta */
11 switch([      ]) {
12   case 1: printf("El promedio es %f\n",
13               [                    ]);
14           break;
15   default: printf("Error: %s\n",
16               [                    ]);
17 }
```

❑ **11.1** Escribe completa la línea 8 que contiene la invocación remota

Respuesta:

El procedimiento remoto tiene el nombre que hemos declarado en el interfaz XDR (`promediar`), más un sufijo indicando el número de versión, que en este caso será `_2`. El parámetro que recibe el procedimiento remoto será `dat`, puesto que es la única variable de tipo `Datos`. Recordemos no obstante que en ONC RPC los parámetros se pasan *siempre* por referencia, por lo que habrá que poner `&dat`. Finalmente, el valor devuelto, según se declara en el interfaz, es de tipo `Respuesta`, pero también se devuelve por referencia por lo que necesitamos una variable de tipo puntero a `Respuesta` que recoja el valor retornado, y que en este caso se trata de la variable `r`.

Solución:

```
r = promediar_2( &dat, conexion );
```

❑ **11.2** Escribe completo el código de la parte que muestra los resultados (línea 11 y siguientes).

Respuesta:

El tipo retornado por el procedimiento remoto es una unión discriminada cuyo discriminante se llama `tipo`. Debemos recordar que en su implementación en C, la unión discriminada se convierte en una estructura con dos campos:

- El discriminante (con el mismo nombre que el declarado en el fichero XDR, en este caso `tipo`)
- Una unión (cuyo nombre es el de la unión declarada en XDR más el sufijo `_u`, en este caso `Respuesta_u`). Esta unión tendrá los campos declarados en el fichero XDR (en este caso dos campos, uno llamado `media`, de tipo `float` y otro llamado `error`, de tipo `char*`, que es la conversión a C del `string`).

Asimismo debemos tener en cuenta que lo que nos devuelve el *stub* del cliente, y que hemos almacenado en la variable `r`, no es una dato de tipo `Respuesta`, sino un *puntero* a ese tipo. Por tanto, para acceder a sus campos debemos usar la sintaxis `r->campo` o bien `(*r).campo`.

Solución:

```
switch(r->tipo) {
  case 1: printf("El promedio es %f\n",
               r->Respuesta_u.media);
          break;
  default: printf("Error: %s\n",
               r->Respuesta_u.error);
}
```

❑ **11.3** En la implementación del servicio, se comprueba si el array recibido tiene elementos. Si es así, se retorna su promedio y si no la cadena `"Sin datos"`. Para ello se usa el tipo `Respuesta` declarado en el fichero xdr. Teniendo en cuenta que la respuesta del servidor va codificada en el formato XDR ¿Cuántos bytes se envían por la red como respuesta en cada caso?

Respuesta:

`Respuesta` es una unión discriminada, de modo que en primer lugar se envía el discriminante (lo que ocupa 4 bytes) y seguidamente el miembro que corresponda. En el caso con datos, el dato retornado en la unión será un `float`, que ocupa otros 4 bytes, haciendo un total de 8 bytes enviados.

En el caso sin datos, el dato retornado en la unión es una cadena con el texto `"Sin datos"`. La codificación XDR de una cadena comienza por un entero que indica su longitud (este entero ocupa 4 bytes y en este caso valdría 00 00 00 09), seguido de tantos bytes como letras tenga la cadena (9 bytes más en este caso), y finalmente ceros de relleno si son necesarios para alcanzar un número de bytes múltiplo de 4 (harían falta 3 bytes de relleno en este caso). El total de bytes será por tanto $4+4+9+3=20$.

Solución:

```
Caso con datos: 8
Caso sin datos: 20
```

Problema 12.

A continuación se muestra una parte del fichero `def.x` que define el interface de una aplicación basada en el ONC RPC de Sun

```
1 program OPERADOR {
2         version OPVERS {
3                 void  ACABA (void) =1;
4                 tipo3 OP1 (int)=2;
5                 int   OP2 (tipo2)=3;
6                 tipo2 OP3 (tipo3)=4;
7         }=5;
8 }=0x2095F999;
```

❑ **12.1** Si en la máquina del cliente tenemos declaradas la siguientes variables

```
1 CLIENT *clnt;
2 int    *dat1, i;
3 tipo2  *dat2;
4 tipo3  *dat3;
```

Escribe el código necesario para hacer una invocación remota al servicio `OP1` en la máquina `serv.com`.

Respuesta:

Se debe inicializar la estructura `clnt` llamando a `clnt_create()`. En esta llamada se le pasa el nombre de la máquina (`"serv.com"`), la constante que identifica al servidor (`OPERADOR`), la constante que especifica el número de versión del servidor (`OPVERS`), y el protocolo de transporte, en el que, ya que no se especifica lo contrario, podemos poner `"tcp"`.

Una vez inicializado `clnt` se puede llamar al servicio. Hay que recordar que la función a invocar se llama como el servicio declarado en el interfaz, pero con un sufijo indicando el número de versión, que en este caso será `_5`. El parámetro que recibe el servicio es un `int`, según el interfaz, pero debe recordarse que en la implementación en C lo que hay que pasarle es un *puntero a ese tipo*, por tanto un `int*`. Ya que la variable `dat1` es precisamente de ese tipo, basta con pasarle esa variable (en este caso no hay que poner el `&` delante, ya que `dat1` *ya es* la dirección de un `int`). El servicio devuelve un `tipo3`, de acuerdo con el interfaz, pero debemos recordar que en la implementación C se recibe un *puntero* a este tipo. La variable `dat3` es por tanto la indicada para recoger este resultado.

Con las consideraciones anteriores, es inmediato completar el código como muestra la solución.

Solución:

```
clnt=clnt_create("serv.com",OPERADOR, OPVERS, "tcp");
dat3=OP1_5(dat1,clnt);
```

❑ **12.2** Si `tipo3` es un array variable de `int`, escribe el código necesario tras la invocación anterior para mostrar el resultado recibido. Por ejemplo, la salida podría ser:

```
Recibidos 3 elementos:
 4  5  9
```

Respuesta:

Recordemos que un array variable se convierte en C en una estructura, cuyos campos se llamarían en este caso `tipo3_len` y `tipo3_val`. La variable `dat3` es por tanto un puntero a una estructura con estos campos.

Para imprimir el número de elementos recibidos debemos acceder al campo `tipo3_len` de la estructura apuntada por `dat3`. Esto se logra con la sintaxis `dat3->tipo3_len` o alternativamente `(*dat3).tipo3_len`.

Para imprimir cada elemento debemos realizar un bucle que se repita tantas veces como indica `dat3->tipo3_len`, y en cada iteración mostrar uno de los elementos apuntados por `dat3->tipo_val`. Podemos usar sintaxis de array con este campo y acceder a `dat3->tipo_val[i]`, o alternativamente `(*dat3).tipo_val[i]`.

Solución:

```
printf("Recibidos %d elementos:\n",
        dat3->tipo3_len);
for (i=0; i<dat3->tipo3_len; i++)
  printf("%d ", dat3->tipo3_val[i]);
printf("\n");
```

Problema 13.

¿Cuál o cuáles de las siguientes afirmaciones son FALSAS?

A) Un *stub* de servidor es una rutina que es llamada desde el código principal del servidor cuando se invoca un procedimiento remoto, pasándosele como parámetro el identificador del servicio.

B) Un interfaz es una declaración de las funciones que ofrece el servidor, en la que se especifican también los tipos de los parámetros y de los valores retornados, y que se escribe en un lenguaje de especificación de interfaces,

C) Para asegurar la semántica de Al menos una vez, el servidor debe almacenar las respuestas que envía a los clientes, para reutilizarlas si es necesario.

D) Las RPC de Sun (ONC RPC) limitan el número de parámetros que un procedimiento remoto puede admitir.

Respuesta:

La afirmación A) es falsa, ya que el *stub* del servidor no es llamado directamente desde el código del servidor, sino que es una rutina a la escucha de clientes, que se ocupa de la traducción de los parámetros y la invocación de la rutina apropiada, información que recibe de la red, y no del código del servidor.

La afirmación C) también es falsa. La necesidad de reutilizar respuestas se da en la semántica de "Como máximo una vez".

Solución:

A, C

Problema 14.

Explica brevemente que es la memoria compartida distribuída y qué ventajas tiene

Solución:

Es un software distribuido que se ocupa de simular la existencia de memoria compartida entre procesos de diferentes máquinas.
Facilita la comunicación entre procesos, pues permite compartir variables, una técnica de uso común entre programadores de arquitecturas paralelas en las que existe memoria compartida real.

Problema 15.

Explica brevemente de qué dos formas se puede usar `fork()` en la programación de un servidor que admita varios clientes

Solución:

Un servidor puede llamar a `fork()` cada vez que acepta un nuevo cliente. El proceso padre, que nunca termina, retornará al `accept()` a esperar por nuevos clientes. El hijo dará servicio al cliente y después terminará con `exit()`.
Otra posibilidad es que un servidor llame a `fork()` varias veces antes de entrar en el bucle de espera por clientes. De este modo todos los hijos ejecutarán el mismo código que el padre, entrando todos a la vez en el `accept()`. Cuando llegue un cliente, solo uno de ellos saldrá del `accept()` y continuará para darle servicio. Cuando haya terminado el servicio, cerrará la conexión con ese cliente y retornará al `accept()` a esperar por otro.

Problema 16.

El siguiente fragmento de código hace algunas llamadas a `fork()` para crear procesos, e imprime algunos mensajes por pantalla

```
1 main()
2 {
3   int i=0, pid;
4   printf("%d, ", i);  i++;
5   pid=fork();
6   printf("%d, ", pid);
7   if (pid==0) i++;  else i--;
8   printf("%d, ", i);
9 }
```

Si el PID del proceso hijo es 16384, escribe una de las posibles secuencias de números que podría leerse en la pantalla al ejecutar el código anterior.

Respuesta:

En el momento que se ejecuta el `fork()` tenemos dos procesos funcionando "a la vez" e intentando ambos imprimir en pantalla. No puede predecirse por tanto en qué orden aparecerán los resultados de la pantalla, por lo que el ejercicio tiene muchas respuestas posibles. Ya que nos pregunta por "una de las posibles", asumamos el caso más simple y supongamos que primero se ejecutan todos los `printf()` del padre y después todos los del hijo.

El primer `printf()` que aparece está antes del `fork()` y simplemente vuelca la variable `i` que en ese momento vale cero. Inmediatamente se incrementa `i` que pasa a valer 1. Posteriormente se ejecuta el `fork()` por lo que, además del proceso actual, aparecerá un segundo proceso "hijo" que heredará el valor de `i` (igual a 1). Consideraremos el hijo más adelante, de momento sigamos con la ejecución del padre.

El padre recibe en la variable `pid` el identificador del hijo, que según el enunciado es 16384 y lo imprime por pantalla. Seguidamente el padre decrementa `i` (pues para el padre no se cumple la condición del `if`) e imprime el nuevo valor, que será cero. Por tanto de momento habría aparecido en pantalla la secuencia `0, 16384, 0`.

Consideremos ahora el hijo. Para él, el valor retornado por `fork()` es cero, y eso será lo que imprimirá el `printf()` de la línea 6. Seguidamente incrementará `i`, ya que para él sí se cumple la condición del `if`, con lo que `i` pasa a valer 2 que será el valor mostrado por el último `printf()`. De modo que a lo impreso por el padre se añade lo impreso por el hijo: `0, 2`.

Una posible respuesta por tanto sería `0, 16384, 0, 0, 2`. Hay otras respuestas posibles, ya que tras el primer cero, los mensajes impresos por padre e hijo pueden intercalarse de múltiples formas. Por ejemplo, otra respuesta válida sería: `0` (antes del `fork()`), `0` (hijo), `16384` (padre), `0` (padre), `2` (hijo).

Solución:

Cualquier intercalación de las dos líneas siguientes:
0, 16384, 0,
0, 2,