Sistemas Operativos Distribuidos
Ejercicios de Examen

Jose Luis Díaz

Extraídos del curso 2005-2006

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 7 y 8 que se incluyen a modo de ejemplo en esta colección.

Problema 1.

Se programa un cliente que conecta con un servidor del servicio "hora" mediante el protocolo TCP. La respuesta del cliente consiste en un entero de 32 bits que representa el número de segundos transcurrido desde 1900. El cliente simplemente imprime este número en pantalla:

```
1  unsigned long int hora;
2  // ... se omite parte ...
3  if (connect(s,&servidor,sizeof(servidor))<0){
4        perror("Connect");    exit(1);
5     }
6  // Recibir el dato
7     [          ]
8     if (n<0){
9        perror("Al recibir"); exit(2);
10    }
11   // Adaptar dato recibido
12    [          ]
13   // Imprimirlo:
14   printf("%lu\n", hora);
15   // Terminar
16   close(s);
```

❑ 1.1 ¿Qué falta en el primer hueco? (línea 7)

Respuesta:

Hay que leer del socket `s` un dato de 32 bits (por tanto 4 bytes) y dejarlo en una variable de tipo entero largo, que será por tanto la variable `hora`.

Solución:

```
n= read(s, &hora, 4);
```

❑ 1.2 ¿Qué falta en el segundo hueco, línea 12?

Respuesta:

El entero recibido está en el orden «de red», que es *big endian*. Sin embargo no sabemos si la máquina en la que lo estamos recibiendo es también *big endian* o por el contrario es *little endian*. Para asegurarnos de que el dato es interpretado correctamente debemos convertirlo al orden de la máquina. Esto lo hace la función `ntohl()`, que es un mnemónico para «*network to host long*», siendo *long* el tamaño del dato. La función recibe un dato que deberá estar en el orden «de red» y devuelve ese dato en el orden «del host», es decir, de la máquina que lo ha ejecutado. Este resultado lo asignamos a la misma variable.

Solución:

```
hora=ntohl(hora);
```

Problema 2.

El siguiente código muestra parte de un cliente UDP para el protocolo echo (puerto 7), programado bajo Windows. La función `InicializarBiblioteca` no se muestra, se asume que inicializa la biblioteca de sockets de Windows. La función `CrearSocketUdp` devuelve un socket UDP ya listo para enviar y recibir. No se hace comprobación de errores a la hora de enviar o recibir datagramas. El código se muestra incompleto y se preguntará por los huecos.

```
1     struct sockaddr_in dir;
2     char *ip=argv[1];
3     char txt[80];
4     int longitud=sizeof(dir);
5     unsigned long int respuesta;
6     InicializarBiblioteca(2,0);
7     sock=CrearSocketUDP();
8     dir.sin_family=[      ];
9     dir.sin_port=[      ];
10    dir.sin_addr.s_addr=[      ];
11 ...
12    printf("Escribe algo:");
13    fgets(txt, 80, stdin);
14    sendto([      ]
15           [      ]
16           [      ]);
17    recvfrom([      ]
```

```
18                [                        ]
19                [              ];
20      printf("El servidor responde: %s\n",
21                txt);
22  ...
```

❑ **2.1** Completa la inicialización de `dir` (líneas 8 a 10

Respuesta:

El campo `sin_family` se rellena siempre con la constante `AF_INET`. El campo `sin_port` se rellena con el número de puerto del servidor (en este caso 37) con cuidado de no olvidar que este número debe almacenarse en orden «de red» y por tanto es necesario convertirlo mediante la función `htons`. Finalmente, el campo `s_addr` debe contener la IP del servidor en un formato de 4 bytes, en orden «de red». En el programa del enunciado observamos que hay una variable de tipo `char *` que se inicializa con el valor de `argv[1]`. Se asume por tanto que el programa recibe como primer parámetro la IP del servidor. Esta IP, no obstante, la tenemos en una cadena de caracteres, y no en el formato requerido para el campo `s_addr`. La función `inet_addr` realiza la transformación necesaria.

Todo lo anterior se programa de forma idéntica en Windows y en Unix.

Solución:

```
dir.sin_family=AF_INET;
dir.sin_port=htons(37);
dir.sin_addr.s_addr=inet_addr(ip);
```

❑ **2.2** Completa la llamada a `sendto()` (línea 14)

Respuesta:

El texto que acaba de ser leído en la variable `txt` debe enviarse al servidor. Para ello se usa la función `sendto` cuyos parámetros han de ser: el socket por donde se envía (`sock`); la dirección de memoria donde está el primer carácter a enviar (dirección del array `txt` que, sintéticamente, se obtiene poniendo tan solo `txt`, lo que equivale a `&txt[0]`); el número de bytes a enviar (que se obtiene averiguando con `strlen()` el número de letras que el usuario ha escrito); las opciones de envío (que son siempre `0`); la dirección del servidor por referencia (es decir, la dirección de memoria donde se almacena la estructura `dir`); y el número de bytes ocupado por esta estructura (que se ha calculado al principio del programa y se ha guardado en la variable `longitud`),

Solución:

```
sendto(sock, txt, strlen(txt), 0 ,
            (struct sockaddr *) &dir, longitud);
```

❑ **2.3** Completa la llamada a `recvfrom()` (línea 17)

Respuesta:

Ahora debe recibirse la respuesta del servidor haciendo uso de la función `recvfrom()`. Los parámetros han de ser: el socket por donde se va a recibir (`sock`), la dirección de memoria donde se dejarán los bytes que envíe el servidor (de nuevo la dirección donde comienza el array `txt`); el número máximo de bytes a recibir (ha de ser el tamaño del array, que se obtiene con `sizeof`); las opciones (de nuevo `0`), la dirección de memoria donde almacenar la estructura `sockaddr_in` con los datos del cliente (si no nos interesan estos datos, podemos pasar `NULL` en este parámetro); y la dirección de memoria donde hay un entero indicando el número de bytes que tenemos disponibles para recibir esta información sobre el cliente (habitualmente se prepara una variable de tipo `int` con el valor `sizeof(struct sockaddr_in)`, pero en este caso, ya que hemos decidido poner `NULL` en el anterior parámetro, no vamos a recibir información alguna, por lo que podemos poner `NULL` también en este último parámetro).

Solución:

```
recvfrom(sock, txt, sizeof(txt), 0,
                    NULL, &longitud);
```

❑ **2.4** Seguidamente se muestra parte del código de la función `CrearSocketUdp()`. Recuerda que se trata de una implementación para Windows.

```
1   [      ] CrearSocketUDP()
2   {
3     [                    ]
4     ret=[                          ]
5     if (ret==INVALID_SOCKET) {
6       perror("Al crear socket");
7       exit(-1);
8     }
9     return ret;
10  }
```

Reescribe las cuatro primeras líneas del código anterior completando lo que falta.

Respuesta:

Hay que recordar que en Windows el tipo de datos para un socket no es `int` como en Unix, sino `SOCKET`. Este debe ser por tanto el tipo retornado por la función (que habrá que completar en la primera línea) y el tipo de la variable `ret` que habrá que declarar al principio de la función. Esta variable `ret` se asigna en la línea siguiente, que debe ser una llamada a la función `socket` para crear el socket. Esta llamada es idéntica a la usada en Unix.

Solución:

```
SOCKET CrearSocketUDP()
{
  SOCKET ret;
  ret=socket(PF_INET, SOCK_DGRAM, 0);
```

Problema 3.

La letra hebrea "aleph" (א) tiene el código Unicode `U+05D0`, y el código `E0` en el estándar ISO-8859-8.

❑ **3.1** Un usuario hebreo configura su editor para usar ISO-8859-8, y escribe dicha letra. El fichero cae en manos de un usuario español que lo abre en Windows ¿Qué carácter verá?

Respuesta:

Verá el carácter cuyo código sea el `E0` en la codificación de Windows. Éste puede averiguarse en la tabla de ISO-8859-1, por ejemplo.

Solución:

à

❑ **3.2** Otro usuario hebreo tenía su editor configurado para usar UTF-8 y escribe la misma letra. ¿Qué secuencia de bytes (hex) se almacena en el fichero?

Respuesta:

El código Unicode del carácter es, como dice el enunciado, `000005D0`. Este código requiere sólo 11 bits para almacenarse (pues los restantes bits a la izquierda son cero). Los 11 bits en cuestión serán 101 1101 0000.

Estamos por tanto ante el caso que usa dos bytes en UTF-8. El primer byte comenzará por la secuencia `110` para indicar que dicha secuencia consta de dos bytes, y el siguiente comenzará por `10` para indicar que es un byte que es parte de una secuencia. Esto deja libres 5 bits en el primer byte y 6 en el segundo, lo que da el total de 11 que se requieren.

Dividiendo los 11 bits de nuestro caso en dos grupos de 5 y 6 bits respectivamente , se obtiene `10111` y `010000`. Anteponiendo el prefijo `110` al primer grupo y `10` al segundo, se obtiene la respuesta `11010111`, `10010000`. Tan solo queda convertirla a hexadecimal.

Solución:

D7, 90

Problema 4.

Un fichero contiene el texto «En un lugar de la Mancha de cuyo nombre no quiero acordarme.» El fichero usa codificación ISO-8859-15 y se decide convertirlo a UTF-8. Al hacerlo, su tamaño ¿aumentará, disminuirá, o quedará igual? Razona la respuesta.

Solución:

Quedará igual debido a que el texto en cuestión sólo contiene caracteres ASCII y la codificación de estos ocupa un solo byte, tanto en ISO-8859-15, como en UTF-8 (además, resulta ser la misma codificación).

Problema 5.

¿Cuál o cuáles de las siguientes afirmaciones son FALSAS?

A) La firma digital de un documento consiste en su cifrado mediante la clave pública del firmante

B) Cuando recibimos una clave pública de alguien debemos pedirle que nos envíe su fingerprint por email para verificar su identidad.

C) Si *A* cifra un documento usando la clave pública de *B*, después ya sólo lo podrán descifrar *A* y *B*.

D) El término «anillo« (ring) se refiere al fichero en que se almacenan todas las claves públicas que hemos aceptado (importado), más la nuestra.

Respuesta:

A) Es falsa, ya que la firma se realiza con la clave secreta en lugar de la pública.

B) Es falsa, ya que el envío del *fingerprint* debe hacerse por un canal seguro, como el telefónico o el papel en mano.

C) Es falsa, ya que una vez cifrado con la clave pública de *B*, sólo se podrá descifrar con la clave secreta de *B*, por lo que ni siquiera *A* podrá descifrarlo.

D) Es verdadera (aunque cabría hacer el matiz de que esa descripción corresponde al anillo de claves públicas).

Solución:

A, B, C

Problema 6.

El siguiente fichero (`tipos.x`) define un tipo XDR

```
1 union Ejemplo switch(int que) {
2       case 1: int x;
3       case 2: string msg<10>;
4       default: float y;
5 }
```

❑ **6.1** Se genera un fichero en formato XDR que contiene el valor de una variable de tipo `Ejemplo`. ¿Cuál es el tamaño máximo que podemos esperar para ese fichero? ¿Cuánto vale el discriminante en ese caso?

Respuesta:

El tamaño de la unión, una vez codificada en XDR, es la suma del tamaño del discriminante (que ocupa siempre 4 bytes) más el tamaño del campo que esté seleccionado por el discriminante (y cuya longitud depende del tipo del campo en cuestión). En nuestro caso, los tres posibles tipos de los campos son `int`, `string` y `float`. El primero y el último ocupan siempre 4 bytes, mientras que el `string` ocupa un número variable de bytes, según el número de letras de la cadena. En este ejemplo la cadena se limita a un máximo de 10 letras, pero incluso con esta limitación, el campo `string` puede ser mayor que los otros dos. Por tanto será para el caso `string` cuando se tenga el mayor tamaño, lo que da respuesta a la segunda pregunta: el discriminante valdrá 2.

En cuanto a la primera pregunta, el tamaño máximo para este caso, se calcularía como sigue: tenemos los cuatro bytes del discriminante, más la codificación del `string`. Ésta se compone de 4 bytes para indicar cuántas letras tiene la cadena, y seguidamente un byte por cada letra. Ya que el máximo es 10, tendremos 10 bytes más, lo que nos da un total de 14 bytes para la cadena. Pero 14 no es múltiplo de 4, por lo que es necesario añadir 2 bytes más de relleno hasta alcanzar 16. El tamaño de la unión será por tanto $4 + 16 = 20$ bytes.

Solución:

```
20 bytes (4 de discriminante, 4 de longitud de cadena, 10 de ca-
dena y 2 de relleno)
Discriminante = 2
```

Una variable de tipo `Ejemplo` es leída de un fichero, y su valor mostrado en pantalla, usando el siguiente código:

```
1  Ejemplo r;
2  FILE *f;
3  XDR op;
4  f=fopen("Datos.xdr", "r");
5  // Inicializar op
6  [              ](                );
7  // Lectura y decodificación de r
8  r.Ejemplo_u.msg=NULL;
9  [              ]
10 // Imprimir resultado
11 if ([                ]) {
12   printf("Mensaje: %s\n",
13               r.[              ]);
14 }
15 ...
```

❑ **6.2** Escribe la inicialización de op (línea 6).

Respuesta:

Para inicializar una variable de tipo `XDR` hay que llamar a la función `xdrstdio_create`, pasándole como parámetros: la dirección de la variable `XDR` a inicializar (en este caso sería la dirección la variable `op`), el fichero en el cual se dejan o toman los datos binarios XDR (en este caso sería `f`), y una constante que indica si se va a codificar o decodificar (en este caso la constante deberá valer `XDR_DECODE`, ya que los datos se leen del fichero).

Solución:

```
xdrstdio_create(&op, f, XDR_DECODE);
```

❑ **6.3** Escribe la lectura y decodificación de `r` en el listado anterior (línea 9)

Respuesta:

Para codificar o decodificar un dato en XDR, basta llamar a la función filtro que lo hace. Esta función siempre tiene un nombre formado por el prefijo `xdr_` y el nombre del tipo que convierte. En este caso se llamaría `xdr_Ejemplo`. Los parámetros siempre son dos: el primero la estructura `XDR` que le dice qué hacer (en este caso es la variable `op`) y el segundo la variable que queremos convertir o en la que queremos obtener el resultado de la conversión (en nuestro caso la variable `r`). Ambos parámetros se pasan *por referencia*, es decir, que tenemos que pasar la dirección de estas variables utilizando el operador `&`.

Solución:

```
xdr_Ejemplo(&op, &r);
```

❑ **6.4** Reescribe el `if` que comienza en la línea 11 del listado anterior, completando las partes ocultas (incluyendo el cuerpo del `if`) .

Respuesta:

Ya que el dato que se lee es una unión discriminada, para imprimir correctamente el valor obtenido es necesario hacer uso del discriminante. Por tanto el `if` estará comprobando si el valor del discriminante es uno concreto, para en ese caso imprimir el campo apropiado. En el cuerpo del `if` vemos que se imprime la cadena `"Mensaje: "` y se hace uso de `%s`, lo que indica que se está accediendo al campo de tipo `string` de la unión. Por tanto la condición del `if` debe ser `if (r.que==2)`, y lo que falta en el `printf` será el acceso al campo de tipo `string`, que ha sido llamado `msg` en el fichero `.x`. Es importante recordar, no obstante, que este campo `msg` no es un campo directo de la variable `r`, sino que forma parte de una unión que es a su vez un campo (llamado `Ejemplo_u`) de la variable `r`. Esto es debido a que las uniones XDR se convierten en C en estructuras con dos campos: el discriminante y la unión originalmente declarada en el `.x`.

Solución:

```
if (r.que==2) {
    printf("Mensaje: %s\n",
           r.Ejemplo_u.msg);
}
```

❑ **6.5** Si los datos que había en el fichero `Datos.xdr` son los siguientes (y en este orden):

```
00 00 00 01 00 00 01 00
```

¿Qué campo de la unión se usó para crearlos y qué valor tenía?

Respuesta:

Los cuatro primeros bytes del fichero han de ser el discriminante. Vemos que el valor es `00000001`, es decir, `1`, y que por tanto se está seleccionando el primer caso de la unión, correspondiente al campo llamado `x` de tipo `int`.

Los siguientes cuatro bytes serán por tanto la codificación de este campo `x`, siguiendo las reglas de codificación de enteros. Vemos que el valor codificado en hexadecimal es `00000100`, que en decimal es 256.

Solución:

El campo `x` que tenía el valor 256, ó 00000100h

Problema 7.

Explica brevemente qué función tiene el *stub* del cliente en el mecanismo de RPC.

Solución:

Se trata de una función que el cliente ejecuta cuando pretende invocar el procedimiento remoto, y que recibe los mismos parámetros que aquel. Su función es localizar la máquina y puerto donde el *stub* del servidor está escuchando, enviarle la petición del servicio que desea ejecutar, enviarle los parámetros (en un formato común de representación de la información) y esperar a recibir la respuesta, decodificar ésta cuando llegue y retornarla como resultado de la función.

Problema 8.

Explica brevemente las similitudes y diferencias entre las codificaciones de caracteres conocidas como UCS-2 y UTF-16

Solución:

Ambas utilizan datos de 16 bits para representar cada carácter, pero UTF-16 puede representar caracteres fuera del plano cero, usando dos datos por carácter y haciendo uso de ciertos códigos especiales (rangos surrogados), mientras que UCS-2 no puede representar caracteres fuera del plano 0 (para UCS-2 los códigos surrogados son simplemente no válidos)