A continuación se recoge una selección de problemas aparecidos en los exámenes del curso 2007-2008. Tras cada pregunta aparece la caja de respuesta, en el mismo tamaño que tenía en el examen, y dentro de ella la solución. Para leerla debes poner la hoja ante un espejo o mirarla por detrás al trasluz. Seguidamente viene la explicación de cómo se llega a esa respuesta, por si tienes dudas.

---

❑ Enumera al menos tres ventajas de un sistema distribuído frente a un computador único.

- Precio: Un conjunto de PCs suele ser más barato que un mainframe con una potencia equivalente.
- Escalabilidad. Si el sistema se queda pequeño, se puede ampliar agregando componentes.
- Más potencia: la potencia de un único computador está limitada.
- Tolerancia a fallos. Si se diseña convenientemente, un fallo en una de sus partes puede no afectar al conjunto.

❑ Explica brevemente qué hace el comando `--sign-key` de `gpg` y cuándo debe usarse.

Este comando añade la firma digital de quien lo ejecute a una clave pública. Esta firma significa que el firmante "certifica" la clave, esto es, da su palabra de haber verificado la identidad del propietario de la clave. Debe hacerse tras verificar la huella dactilar (*fingerprint*) de la clave que se firma, asegurándose de que esta huella se recibe por canal seguro.

❑ El siguiente código muestra parte de un **servidor** para servicio `hora` bajo el protocolo UDP. El servidor, cuando recibe una petición, averigua la hora que es consultando la función `time` del Sistema Operativo. Esta función retorna un contador de segundos desde el 1 de enero de 1970. Seguidamente el servidor ajusta esta hora para que se amolde al estándar del protocolo `time` y lo envía al cliente, en formato *big endian*. El código tiene partes tapadas sobre las que se harán preguntas. No se hace comprobación de errores.

```
1  #include<time.h>
2  #include<socket.h>
3  // ... más includes que se omiten
4
5  // Definimos una constante que contiene el número
6  // de segundos transcurridos desde 1900 hasta 1970
7  #define OFFSET 2208988800LU
8  #define PUERTO_TIME 37
9
10 main()
11 {
12   int sock;
13   time_t hora;       // time_t es un entero
14   struct sockaddr_in serv, client;
15   unsigned long int lserv=sizeof(serv),
16                     lclient=sizeof(client);
17   long int aux;      // La hora a enviar al cliente
18
19  // Preparar socket de servicio
20   sock=socket(PF_INET, SOCK_DGRAM, 0);
21   serv.sin_family=AF_INET;
22   serv.sin_addr.s_addr=htonl(INADDR_ANY);
23   serv.sin_port=htons(PUERTO_TIME);
24   [                                    ]
25
26 while(1) {
27   // Esperar clientes
28   [                                                    ]
29   // Averiguar la hora
30   hora=time(NULL);
31
32  // Corregirla para que siga el estandar
33   [                    ]
34   [                    ]
35   // Enviarla al cliente
36   [                                                    ]
37   } // Fin while
38 }// Fin main
```

— ¿Qué falta en la línea 24?

```
bind(sock, &serv, lserv);
```

**Explicación:** Es necesario asignar un puerto al socket, por lo que falta una llamada a la función `bind`, con los parámetros apropiados. Al ser un socket UDP, no se debe llamar a `listen`.

— ¿Qué falta en la línea 28?

```
recvfrom(sock, &aux, 0, 0,
         &client, &lclient);
```

**Explicación:** La espera de clientes, en un socket UDP, es lo mismo que esperar a que llegue un datagrama. Por tanto debemos llamar a `recvfrom` sobre el socket `sock`. En el caso del protocolo `time`, el datagrama que esperamos es vacío, por lo que en el tercer parámetro debemos poner `0`. El segundo parámetro (dirección donde se almacenará el datagrama) puede ser la dirección de cualquier variable, ya que en realidad no se recibirá nada. El cuarto parámetro son opciones de recepción, que no necesitamos, y por tanto pondremos `0`. Los dos últimos parámetros en cambio son fundamentales, ya que a través de ellos recibiremos la dirección del cliente que nos ha enviado el datagrama, que es el cliente a quien debemos responder con la hora. Almacenaremos la dirección del cliente en la estructura `client`, y su longitud en `lclient`, y por tanto pasaremos las direcciones de estas variables como quinto y sexto parámetros.

— ¿Qué falta en la línea 33 y la siguiente? (Nota, también puede realizarse la operación en una sola sentencia)

```
aux=htonl(aux);
aux=hora+OFFSET;
```

**Explicación:** En estas líneas estamos preparando la respuesta que se enviará al cliente. Esta respuesta debe ser la hora (obtenida en la línea anterior con `time`), pero con origen de tiempos en 1900 en lugar de 1970, y ordenada en el orden de red. Por tanto, la preparación consiste en sumarle el `OFFSET` declarado como constante al principio del listado, y ajustar la *endianity* del dato con la macro `htonl` (observar que en este caso estamos implementando un servidor, no un cliente, y que la hora va a ser enviada, por lo que hay que convertirla del orden de *host* al orden de red).
Preparamos el resultado a enviar en la variable `aux`, aunque también podría hacerse todo sobre la misma variable `hora`.

— ¿Qué falta en la línea 36?

```
sendto(sock, &aux, 4, 0,
       &client, lclient);
```

**Explicación:** Se trata simplemente de enviar la respuesta que hemos preparado en la pregunta anterior. Usamos la función `sendto` para enviar un datagrama compuesto por los 4 bytes que hay en la variable `aux`. Se envía a la dirección contenida en `client` (que es la que obtuvimos en el `recvfrom`)

— ¿Por qué no aparece ninguna llamada a `close()` dentro del bucle `while`?

Porque se trata de un servidor UDP, por tanto orientado a datagramas y sin conexión. No hay un socket de escucha y otro de datos separado, por lo que no se cierra el de datos una vez terminamos con el cliente para seguir con el de escucha abierto. Tampoco podemos cerrar el único socket que tenemos pues no podríamos recibir más clientes.

❑ El fichero `tipos.x` contiene las declaraciones siguientes:

```
typedef string Texto<18>;
struct Complejo {
  float re;
  float im;
};
struct ListaComplejos {
   Complejo x;
   ListaComplejos *siguiente;
};
union Result switch(int caso) {
  case 1: Complejo r;
  case 2: Texto err;
  default: int codigo;
};
```

— ¿Cuántos bytes como máximo ocupará un dato del tipo `Result` codificado en XDR? ¿Y cuántos como mínimo?

MÁX: 28 MÍN:8

**Explicación:** La unión ocupa 4 bytes para el discriminante, más el tamaño de la rama elegida.
La rama que más puede llegar a ocupar es la del caso 2, ya que en el caso 1 el tamaño sería el de un `Complejo` (que son 8 bytes) y en el caso *default* sería el de un entero (4). En cambio en el caso 2 tenemos un `Texto`, que puede llegar a tener hasta 18 caracteres, de acuerdo con la definición del tipo `Texto`.
Este caso ocupará 4 bytes para indicar la longitud del texto más lo que ocupe el texto en sí. Como máximo es 18, por tanto 4+18=22. Como no es múltiplo de 4 se redondea a 24. Sumando el tamaño del discriminante obtenemos que el tamaño máximo de esa unión sería de 28 bytes.
Para el caso mínimo tendremos de nuevo los 4 bytes del discriminante más lo que ocupe la rama. La rama *default* ocupa siempre 4 bytes (el tamaño del `int`). La rama `Texto` tiene tamaño variable, pero en el caso mínimo, el texto podría ser una cadena de longitud cero. En todo caso, se requieren 4 bytes para almacenar el entero que contiene la longitud de la cadena. Así pues, el caso mínimo se produce para el caso 2 con una cadena vacía, o para el caso *default*. En ambos casos el tamaño sería 8 (4 del discriminante y 4 de la rama).

— La estructura `Complejo` almacena en cada campo `re`, `im`, la parte real e imaginaria respectivamente de un número complejo. Supongamos que se crea una variable de tipo `ListaComplejos` de modo que la lista contenga dos elementos. El primero sería el número real 1,0 (parte imaginaria cero), y el segundo el número imaginario $1{,}0i$ (parte real cero). Escribe cuáles serían los bytes que resultan de la codificación en XDR de dicha lista. Escribe un byte en cada casilla. Pueden sobrar casillas. **Nota:** La codificación del número 1,0 en la norma IEEE-754 es `3F800000`, y la codificación del 0,0 es `00000000`.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3F | 80 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00 | 00 | 00 | 01 | 00 | 00 | 00 | 00 |
| 3F | 80 | 00 | 00 | 00 | 00 | 00 | 00 |

**Explicación:** La lista se codifica campo a campo. El primer campo es de tipo Complejos, el cual al ser una estructura también se codifica campo a campo (primero el campo `re` y después el campo `im`). El segundo campo es un dato opcional y por tanto se codifica como `00000001` si está presente (esto es, si hay más elementos en la lista) o como `00000000` si no lo está (esto es, si es el último dato de la lista).
El primer elemento de la lista, por tanto, se codificará como el campo `re`, seguido del campo `im` del primer número complejo, y después el dato `00000001` para indicar que viene otro elemento. El segundo elemento se codificará como el campo `re` seguido del campo `im` del segundo número complejo, y después el dato `00000000` para indicar que ya no hay más elementos. En cuanto a los campos `re` e `im`, para el primer elemento son 1,0 y 0,0 respectivamente, y para el segundo elemento son 0,0 y 1,0. Puesto que el problema ya nos dice cómo se codifican estos números en IEEE-754, tenemos toda la información necesaria y sólo queda rellenar los bytes en cuestión como se muestra en la respuesta.

— Escribe cómo es el equivalente en C generado por `rpcgen` del tipo de datos `Result`

```
struct Result {
    int caso;
     union {
  Complejo r;
   Texto err;
    int codigo;
  } Result_u;
};
typedef struct Result Result;
```

— Escribe las instrucciones necesarias para declarar una variable llamada `r` del tipo `Resp` y para asignarle el mensaje de error `"Resultado no definido"`

```
Respuesta r;
r.caso=2;
r.Result_u.err="Resultado no definido";
```

**Explicación:** El mensaje de error debe ir en el campo de tipo `Texto`, por lo que el discriminante debe ser inicializado con el valor 2. Para asignar el mensaje, no debemos olvidar que es un campo dentro de una unión interna de nombre `Result_u`. Podemos asignar el string directamente o haciendo uso de funciones de manejo de cadenas, como `strlen()`, `strdup()`, o `strcpy()`

El siguiente fragmento de código tiene por cometido recibir desde la máquina `"gollum"` una variable de tipo `Result` e imprimir su valor por pantalla. El listado tiene unos huecos por los que se te preguntará a continuación.

```
1  ...
2  int sock,s;
3  struct sockaddr_in serv;
4  FILE *f;
5  XDR op_leer;
6  Result z;
7
8  sock=socket(PF_INET, SOCK_STREAM, 0);
9  /* Inicialización de estructura serv omitida */
10
11 s=connect(sock, &serv, sizeof(serv));
12 f=fdopen(                    );
13 xdrstdio_create(                                  );
14 memset(&z, 0, sizeof(z));
15 /* Recibir el dato */
16 if (                                !=TRUE) {
17   fprintf(stderr, "Error al recibir\n");
18   exit(-1);
19 }
20 /* Imprimir el valor de z */
21 switch(z.caso) {
```

```
22 case 1:  printf("Resultado complejo: %f+%fi\n",
23                 [redacted], [redacted]);
24          break;
25 case 2:  printf("Error: %s\n",
26                 [redacted]);
27          break;
28 default: printf("Código: %d\n",
29                 [redacted]);
30 }
```

— ¿Qué falta en las líneas 12 y 13?

```
f=fdopen(s, "r");
xdrstdio_create(&op_leer, f, XDR_DECODE);
```

**Explicación:** El socket que ha sido creado y conectado, debe ser convertido al tipo `FILE` para que los filtros XDR puedan manejarlo. Esto lo hace la función `fdopen` a la que se le pasa el descriptor del socket (`sock`) y el modo de apertura, en este caso `"r"`, puesto que vamos a leer de él. Una vez tenemos el resultado de tipo `FILE` en la variable `f`, usaremos la función `xdrstdio_create` para inicializar una estructura de tipo `XDR` (variable `op_leer`) asociándola al fichero `f` y la operación `XDR_DECODE`.

— Completa la línea 16

```
} (xdr_Resultf(&op_leer, &z)!=TRUE)
if
```

**Explicación:** Como se indica en el comentario anterior al hueco, lo que sigue tiene por cometido recibir el dato. Puesto que estamos trabajando con XDR, para recibir un dato todo lo que hay que hacer es llamar al filtro para ese tipo de dato. En este caso, llamaremos a `xdr_Result`. El filtro recibe como primer parámetro la dirección de una variable de tipo `XDR` que le indica de dónde leer (en este caso la variable será `op_leer` que le indica que debe leer del fichero `f`, que a su vez representa al socket `sock`), y la dirección de otra variable donde dejar lo que lee (en este caso la variable es `z`, de tipo `Result`). El filtro devuelve `TRUE` si todo va bien. En caso contrario, como vemos, el programa imprime un error y finaliza.

— Completa el `printf` de la línea 23

```
z.Resultf_u.r.re, z.Resultf_u.r.im
```

**Explicación:** En el caso 1, el resultado es de tipo `Complejo`. Vemos que el `printf` muestra el número comlejo en la forma `%f+%fi` (por ejemplo `0.0+1.0i`) y por tanto hay que pasarle dos `float` (parte real e imaginaria del resultado). Esos son los campos `re` e `im` de la rama llamada `r` en la unión. Recordemos que a su vez esta unión es un campo llamado `Result_u` en la variable `z`. Es decir, `z.Result_u` es la unión interna, `z.Result_u.r` es el campo que hay que acceder en el caso 1, y al tratarse a su vez de una estructura de tipo `Complejo`, se tienen `z.Result_u.r.re` y `z.Result_u.r.im` para acceder a sus campos, que son los que espera `printf`.

— Completa los `printf` de las líneas 26 y 29.

```
z.Resultf_u.error
z.Resultf_u.codigo
```

**Explicación:** Véase la explicación de la pregunta anterior. En este caso es análoga, pero accediendo a los campos apropiados de la unión, según el caso del `switch`.

❑ ¿Cuál o cuáles de las siguientes afirmaciones son ciertas?

**A)** La codificación de caracteres utilizada por Windows es compatible con el estándar ISO-8859-15 excepto por los códigos en el rango `0x80` a `0x9F`

**B)** Cuando se codifica un fichero en UTF-16 es necesario introducir el carácter denominado *byte order mark* al principio del mismo.

**C)** La codificación UTF-16 mantiene la compatibilidad con los *strings* del lenguaje C.

**D)** Un texto escrito en español, codificado como UTF-8 ocupará ligeramente más que el mismo texto codificado con la página de códigos de Windows (cp1251).

```
B, D
```

**Explicación:** A) Es falsa. La codificación de Windows es compatible con ISO-8859-1, no con ISO-8859-15.

B) Es cierta. Ese carácter tiene por código `FEFF` y según sea almacenado como `FE, FF` o como `FF, FE` se podrá deducir la *endianity* de la máquina que lo generó y así reconstruir correctamente los caracteres.

C) Es falsa. Una cadena en C usa un byte nulo como terminador. Una cadena UTF-16 contendrá generalmente muchos caracteres nulos en su interior, debido a que un ASCII codificado en 16 bits tiene el byte alto a cero.

D) Es cierta. El texto español es en su mayor parte ASCII, cuya codificación ocupa 1 byte también en UTF-8, pero las letras acentuadas o la eñe, en la tabla de códigos 1251 ocupan 1 byte, mientras que en UTF-8 ocupan dos.

❑ Se ejecuta la siguiente línea de código de un programa en C:

```
int len;
len = strlen("Piña:2€");
```

— Si el programa anterior se ha escrito desde un editor que guarda el texto codificado según el estándar ISO-8859-15 ¿qué valor quedará asignado a la variable `len`? ¿Y si el editor guardara en UTF-8? ¿Y en UTF-16LE?

ISO: len=7 UTF-8: len=10 UTF-16LE: len=1

**Explicación:** En realidad `strlen` realiza la cuenta del número de bytes (que no caracteres) que contiene la cadena hasta la aparición del primer byte con valor 0. Este byte de valor cero lo añade el compilador C al cierre de las comillas. En la codificación ISO cada letra ocupa 1 byte, por lo que en este caso `strlen` devolverá 7, que es el número de caracteres del texto “Piña:2€”.

En la codificación UTF-8 algunos caracteres ocupan 1 byte, otros 2, otros 3 y otros 4. Depende de su código Unicode. En particular, todos los ASCII ocupan 1 byte, los presentes en la tabla ISO-8859-1 dos bytes, y el resto de caracteres del plano 0, 3 bytes. En la cadena en cuestión casi todo son caracteres ASCII, excepto la ’ñ’ (2 bytes pues está en ISO-8859-1) y el ’€’ (3 bytes pues NO está en ISO-8859-1). El total de bytes es por tanto 10, que será el valor retornado por `strlen`

Finalmente en UTF-16 cada carácter ocupa 2 bytes, pero en el caso particular “LE” (*Little-Endian*) el primero de ellos es la parte baja del código y el segundo la parte alta. La primera letra del texto es ’P’, que es ASCII y por tanto tendrá en su parte alta `00`. Esto implica que el segundo byte de la cadena es `00`, lo que causará que `strlen` deje de contar, al tomarlo por el terminador. Por tanto sólo contará 1 byte y ese será el valor asignado a `len`.

— Escribe los bytes que resultan de codificar en UTF-8 la cadena anterior "Piña:2€". (La tabla contiene más huecos de los necesarios, rellena sólo hasta donde sea preciso)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 50 | 69 | c3 | b1 | 61 | 3a | 32 | e2 |
| 82 | ac | 00 | | | | | |

**Explicación:** La mayor parte de los caracteres de la cadena son ASCII, y por tanto su codificación utf8 coincide con la de la tabla ASCII. Los únicos que se apartan de este patrón son los caracteres ’ñ’ y ’€’. Podemos encontrar ambos caracteres en la tabla correspodiente al estándar ISO-9959-15, donde podemos leer su código Unicode, que resulta ser U+00F1 para la ’ñ’ y U+20AC para el ’€’.

El primero de ellos requiere dos bytes para codificarse, que responderán por tanto al patrón `110xxxxx`, `10xxxxxx`, rellenándose las

x con los últimos 11 bits del código (00F1=00011110001), resultando por tanto 110 00011 y 10 110001, es decir, C3 y B1 en hexadecimal.

El segundo de ellos requiere tres bytes, que responderán por tanto al patrón 1110xxxx, 10xxxxxx y 10xxxxxx. Como vemos hay 16 bits a rellenar, que son los 16 bits del código (20AC=0010000010101100), resultando por tanto 1110 0010, 10 000010 y 10 101100, es decir, E2, 82 y AC en hexadecimal.

Finalmente, la cadena tiene un terminador nulo (byte igual a cero).

❑ Se implementa un servidor del protocolo `echo` capaz de manejar varios clientes simultáneos por el método de crear procesos (`fork()`) cada vez que llega un cliente, mientras que el padre permanece a la espera de clientes nuevos. El programa se ha implementado de forma incorrecta, debido a que el padre no maneja la señal `SIGCHLD`. Desde que el servidor ha arrancado, hasta el instante actual, se han conectado 4 clientes, y desconectado dos de ellos.

¿Cuántos procesos habrá en ejecución en el instante actual? ¿Cuántos zombies?

3 en ejecución mas 2 zombies (que no están en ejecución)

**Explicación:** En ejecución está inicialmente el proceso padre. Cada vez que llega un cliente, se lanza un proceso hijo para atenderle. Cuando el cliente desconecta, el proceso hijo termina su ejecución y, debido a la incorrecta implementación del padre, queda en estado *zombie*. Ya que han llegado 4 clientes, se habrán creado 4 nuevos procesos (más el padre). Y ya que dos de ellos han desconectado, 2 de estos procesos finalizan, por lo que sólo quedan en ejecución 3 (2 atendiendo a los clientes y el padre en espera de nuevos clientes). Los dos que han finalizado han quedado en estado *zombie*.

❑ ¿Cuál o cuáles de las siguientes afirmaciones es FALSA?:

**A)** En la criptografía de clave pública no es necesario compartir la clave secreta en los dos extremos de la comunicación.

**B)** Los sistemas distribuidos son más inseguros que los centralizados.

**C)** El ticket que utiliza Kerberos permite autenticar a un cliente por un tiempo indefinido.

**D)** En RSA, N se calcula como P*Q y Z como (P-1)*(Q-1)

C

❑ El operador $\rightarrow$ denota la precedencia lógica y nombramos los eventos de forma que dos eventos con la misma letra ocurren en la misma máquina y el subíndice del evento indica que $A_i \rightarrow A_j$ siempre que $i < j$.

Sabemos además que $C_1 \rightarrow A_2$ y $C_2 \rightarrow B_3$ ¿Qué relación podemos afirmar entre $A_2$ y $B_3$?

$A_2 || B_3$

**Explicación:** Ya que $C_1 \rightarrow C_2$, se deduce que $C_1 \rightarrow B_3$. Por tanto tenemos que $C_1 \rightarrow A_2$ y que $C_1 \rightarrow B_3$. No obstante es imposible saber entre $A_2$ y $B_3$ cuál es anterior. Sólo cabe decir que son concurrentes.

❑ El siguiente código pretende poner en hora el reloj de un computador que forma parte de un sistema distribuído. Para ello hace uso de las siguientes funciones:

- `time(NULL)` Esta función devuelve el contador de segundos de la máquina local, con origen de tiempos en el 1 de Enero de 1970.
- `servicio_time(maquina)` Esta función le pide la hora a la máquina que recibe como parámetro, usando el protocolo `time`. La función devuelve el valor respondido por la máquina (ya con el orden de bytes correcto), que tiene como origen de tiempos el 1 de Enero de 1900.
- `poner_hora(contador)` Cambia la hora local de la máquina, de forma que la nueva hora sea la especificada en `contador`, que debe tener origen de tiempos en 1970. En esta implementación no nos preocupamos por evitar cambios bruscos o retrocesos en dicho reloj local.

```
1 #define OFFSET 2208988800LU
2 long int hora1, hora2, lahora;
3 hora1  = time(NULL);
4 lahora = servicio_time("hora.uniovi.es");
5 hora2  = time(NULL);
6 poner_hora(                                  );
```

Para estimar la hora correcta a partir de la respuesta del servidor, utiliza el algoritmo de Christian. ¿Qué falta entonces en el hueco?

lahora - OFFSET + (hora2 - hora1)/2

**Explicación:** El algoritmo de Christian trata de tener en cuenta los retardos en la red. Es decir, entre el instante en que se pide la hora (instante recogido en `hora1` y el instante en que se recibe (recogido en `hora2`), ha pasado un tiempo debido al retardo de la red. La respuesta que envió el servidor (recogida en `lahora`) está ya anticuada cuando llega al cliente, pues ya es más tarde. Pero ¿cuánto más tarde? El algoritmo de Christian consiste en suponer que la respuesta del servidor se originó en el punto medio entre `hora1` y `hora2`. Por tanto cuando se recibe la hora, la hora real es la que marcaba el servidor más `(hora2-hora1)/2`.

Aparte del ajuste anterior debido al algoritmo de Christian, es necesario otro ajuste debido al diferente origen de tiempos que usa el protocolo time. La respuesta del servidor está medida desde 1900, pero los restantes tiempos que se manejan van medidos desde 1970. Es necesario restar a la respuesta del servidor el "número mágico" recogido en la constante `OFFSET` (que es la cuenta de segundos transcurridos entre 1900 y 1970).