UNIVERSIDAD CARLOS III DE MADRID
AREA DE ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES
GRADO EN INGENIERÍA INFORMÁTICA. SISTEMAS DISTRIBUIDOS

Para la realización del presente examen se dispondrá de **2 horas y 30 minutos. NO** se podrán utilizar libros, apuntes **ni** calculadoras de ningún tipo. **Responda** a los ejercicios en el **espacio reservado**.

Alumno: _____ **Grupo:** _____

Pregunta 1 (2 puntos): Un cliente envía una petición de longitud L bytes a un servidor con un único procesador. El servidor procesa la petición en un tiempo t y devuelve una respuesta de longitud L/2 bytes. El tiempo t de procesamiento de una petición sólo supone tiempo de CPU. La red dispone de un ancho de banda de 1Mbit/s. Responda razonadamente a las siguientes preguntas:

a) ¿Cuál es el tiempo de respuesta percibido por el cliente?
b) Si el servidor es secuencial, ¿cuántas peticiones como máximo podría atender en el intervalo de tiempo [1,N]?
c) ¿Y si el servidor es concurrente?
d) Asuma que el servidor dispone de 4 procesadores, ¿cuál sería el tiempo percibido por el cliente?
e) Ventajas e inconvenientes de un servidor secuencial frente a un servidor concurrente.

Solución:

a) El tiempo de respuesta que el cliente percibe es el tiempo de transferencia del mensaje de petición, más el tiempo de cómputo más el tiempo de transferencia del mensaje de respuesta, es decir:

$$T_{total} = \frac{1}{Mbps}(8L + 4L) + t = 12L * 10^{-6} + t$$

b) Con un servidor secuencial, en tiempo N se podrían atender como máximo n peticiones:

$$n = \frac{N}{Ttotal}$$

NOTA: Asumimos también que no se encolan los mensajes de petición por lo que no se puede paralelizar la comunicación con el cómputo. En el caso de que la comunicación se pueda solapar con el cómputo sería:

$$n = \frac{N}{t}$$

c) Si el servidor es concurrente, entonces se podría paralelizar el cómputo de tantas peticiones como procesadores tenga el servidor. En este caso, dado que tenemos 1 único procesador la concurrencia es virtual y no real, por tanto n es igual que en el caso b).

d) En este caso se podría paralelizar el cómputo de tantas peticiones como procesadores tenga el servidor, es decir, 4. Asumiendo que la comunicación se puede paralelizar con el cómputo tenemos:

$$n' = n * 4$$

e) Ventajas:
- Un servidor secuencial es más sencillo de programar, dado que no tiene en cuenta cuestiones de programación concurrente (gestión de procesos ligeros, acceso a variables compartidas, etc.).

Inconvenientes:
- El rendimiento es peor comparado con el servidor concurrente, dado que los tiempos de comunicación y cómputo no pueden ser paralelizados. Esto implica que el tiempo de respuesta que los clientes perciben es mayor, o lo que es lo mismo, en un instante de tiempo un servidor sólo puede procesar 1 petición.

Pregunta 2 (1 punto): Responda a las siguientes cuestiones:

a) En un sistema basado en RPC ¿qué quiere decir que el *binding* es persistente? Ventajas e inconvenientes de un *binding* persistente frente a uno no persistente.
b) ¿En qué consiste el paradigma de computación peer-to-peer? Indique tres ejemplos de aplicaciones basadas en este paradigma.

Solución:

a) Un *binding* persistente quiere decir que el enlace dinámico entre el cliente y el servidor de RPC se realiza una única vez, es decir, el mismo enlace se mantiene entre múltiples peticiones de RPC mientras que en un *binding* no persistente es necesario establecer el enlace en cada petición de RPC.
   Ventajas del *binding* persistente:
   - Más eficiente: la localización del servidor sólo se realiza una única vez, antes de la primera petición de RPC, por tanto se necesitan menos mensajes lo que redunda en menor tráfico.
   - Útil cuando hay muchas peticiones de RPCs.

   Inconvenientes:
   - Menos tolerante a fallos: si el servidor de RPCs falla, el cliente no será capaz de detectar el error puesto que el enlace ya se ha establecido y es persistente.
   - No permite la migración de servicios.

b) El paradigma de computación peer-to-peer permite la comunicación entre pares de procesos a los que no se asigna un rol de cliente o de servidor. A diferencia del modelo cliente/servidor donde el cliente inicia siempre la comunicación, en este paradigma cualquiera de los procesos puede iniciar una comunicación con otro proceso y puede actuar al mismo tiempo como cliente y servidor. Ejemplos de aplicaciones peer-to-peer: Napster, BitTorrent, Spotify.

Pregunta 3 (1,5 puntos): Se disponen de 3 clientes A, B y C que usan tres sistemas de ficheros distribuidos SF1, SF2 y SF3, cada uno de ellos empleando un modelo de acceso diferente. Un cliente A usa el SF1 que emplea el modelo de carga/descarga, un cliente B usa el SF2 que emplea acceso remoto (sin caché en los clientes) y un cliente C que usa el SF3 que emplea un modelo híbrido. Los clientes A, B y C quieren acceder para lectura a un único bloque del fichero 'foo.txt'. Complete la siguiente tabla:

	Cliente A	Cliente B	Cliente C
Servicios del SFdel cliente que implican comunicación con S	open	open, read y close	open, read y close
Unidad de transferencia C→S	fichero	--	--
Dónde se realiza la operación	cliente	servidor	servidor
Unidad de transferencia S→C	fichero	tamaño de la petición	bloque
Número de mensajes de petición/respuesta	2 (transferencia del fichero completo del C→S y S→C)	3 peticiones cliente (open, read y close) + 1 respuesta servidor (petición)	3 peticiones cliente (open, read y close) + 1 respuesta servidor (bloque)

Notación: S: Servidor/C: Cliente

Solución:

- La unidad de transferencia entre cliente y servidor es el fichero en el caso del modelo carga/descarga. En el caso del modelo de acceso remoto se transfiere el tamaño de la petición solicitada y en el caso del modelo híbrido, el servidor devuelve al cliente el bloque solicitado. Una vez enviada la respuesta en los dos últimos modelos, el cliente no devuelve nada al servidor.

- El número de mensajes de petición respuesta tiene en cuenta todos los mensajes de petición de los servicios del sistema de ficheros

Pregunta 4 (1,5 puntos): Considere un sistema distribuido compuesto por N=5 nodos que mantienen réplicas de acuerdo al algoritmo de votación dinámica (quórum). Inicialmente:

	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5
SC	5	5	5	5	5
NV	8	8	8	8	8

La red se fragmenta en dos particiones, una partición 1 compuesta por {1,5} y una partición 2 compuesta por {2,3,4}. Se pide representar el valor de SC y NV después de cada uno de los siguientes eventos:

a) Actualización en la partición 2.

	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5
SC	5	3	3	3	5
NV	8	9	9	9	8

b) La partición 2 se fragmenta en dos particiones más, formadas por los nodos {2} y {3,4}. Actualización en la partición {3,4}

	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5
SC	5	3	2	2	5
NV	8	9	10	10	8

Pregunta 5 (4 puntos): Se desea diseñar un servicio de *broadcast* en el que un proceso envía un mensaje a todos los N procesos de un sistema distribuido excepto a sí mismo. En este sistema, el identificador de los procesos oscila entre 0 y N-1 y es conocido por todos los procesos. Los procesos ejecutan en la misma máquina pero en puertos distintos. Se asumirá que la dirección IP de la máquina es 168.222.12.12 y que los procesos usan puertos consecutivos a partir del puerto 10000 (el proceso con ID igual a 0 usa el puerto 10000, el proceso con ID igual a 1 usa el 10001, etc.).

El servicio debe proporcionar cierta forma de fiabilidad, por lo que se dispone de un servicio, denominado `timeout`, que permite gestionar un temporizador de retransmisiones. El servicio `timeout` bloquea al proceso t unidades de tiempo o hasta que un mensaje es recibido. Devuelve un 1 si el temporizador ha expirado y 0 en caso contrario. Su prototipo es el siguiente:

- `int timeout(int t);`

El servicio de `broadcast` tiene el siguiente prototipo:

- `int broadcast(int N, char *msg,int t);`

donde `N` es el número de procesos a los que se envía el mensaje, `msg` es el mensaje a enviar y `t` es el tiempo de espera (en segundos).

Se dispone además de la siguiente interfaz de paso de mensajes:

- `send(i,msg,size)` – envia el mensaje `msg` de tamaño `size` al proceso `i`
- `receive(i,&msg,size)` – recibe el mensaje `msg` de tamaño `size` del proceso `i`

Se pide:

a) Escribir el pseudocódigo que permite conectar a todos los procesos entre sí.
b) Escribir el pseudocódigo de la función broadcast asumiendo que todos los procesos están conectados entre sí.
c) Escribir el pseudocódigo de una función que implemente un broadcast fiable.
d) Se quiere implementar la función broadcast usando el API de sockets de C.
    1. Describir las cuestiones de diseño necesarias para realizar una implementación con sockets que cumpla los requisitos anteriores.
    2. Implementar el código que permite conectar a todos los procesos entre sí en C.

3. Implementar la función `broadcast` en C.

Solución:

a) La función de conexión de los procesos permite el establecimiento de la conexión entre el proceso que ejecuta broadcast y el resto de procesos.

Este problema admite varias soluciones. Se selecciona la solución en la que el proceso que realiza el broadcast establece la conexión con el resto de procesos, los cuales a su vez deben aceptar la conexión. Una vez realizado el envío se cierra la conexión.

El seudocódigo sería el siguiente:

```
/* Variables globales */
struct sockaddr_in direcciones[N];
int conexiones[N];
int puerto= 10000;

int conectar_procesos(int mi_id){
	int s;
	int i;
	int ret=-1;

	// crear las direcciones de los procesos receptores
	for(i=0;i<N;i++){
		direcciones[i].IP="168.222.12.12";   /* En pseudocodigo */
		direcciones[i].puerto=puerto+i;
	}

	// establecer la conexión
	i=0;
	while(i<N){
		// creo un socket
		if (i!=mi_id){
			conexiones[i]=crear_socket();
			ret=-1;
			while(ret!=0)
			ret=conectar(conexiones[i],direcciones[i],sizeof(direcciones[i]));
			i++;
		} /* if */
	}/* while */
}
```

b) Se asume que los procesos están conectados ya, y que para el envío solo se necesita conocer el identificador del proceso respetando la interfaz de send y receive.

```
int broadcast(int N, char *msg, int t){
	struct message respuesta;

	for(i=0;i<N;i++){
		if (mi_id!=i){
			send(i, msg, sizeof(struct message))
			receive(i, &respuesta, sizeof(struct message));
		}
	}
}
```

c) El broadcast fiable implementa reenvíos en el send y el timeout en el receive:

```
int broadcast_fiable(int N, char *msg, int t){
	struct message respuesta;
	int ret;

	for(i=0;i<N;i++){
		if (mi_id!=i){
			recibido=0;
			while(!recibido){
				n=0;
```

```
                                len=sizeof(struct message);

                                /* Envío fiable */
                                while(n<sizeof(struct message){
                                        n=send(i, msg, len);
                                        len=len-n;
                                        msg=msg+n;
                                }
                                ret=timeout(t);
                                if (ret==0){
                                        receive(i, &respuesta, sizeof(struct message));
                                        recibido=1;
                                }
                        }
                }
        }
```

d.1 Cuestiones de diseño.

- **Nombrado de los procesos**: necesitamos saber cómo identificar los procesos receptores del broadcast y además el proceso que realiza el broadcast. Todos los procesos ejecutan en la misma máquina, por tanto la dirección IP es la misma (168.222.12.12) y usan un número de puerto distinto que se asigna de la siguiente manera: el proceso i ejecuta en el puerto p+i.

- **Heterogeneidad:** dado que las máquinas cliente y servidor pueden tener arquitecturas de hardware distintas, para realizar el envío de los datos a la red deben traducirse al estándar de red (*network order o big endian*); análogamente en la recepción los datos desde la red deben pasarse al formato de host. Estas dos operaciones se denominan *marshalling* y *unmarshalling* respectivamente.

- El **protocolo de transporte** a utilizar va a depender de los requisitos de fiabilidad de la aplicación. Los protocolos de transporte implementados sobre IP son TCP y UDP. En este caso dado que nos piden implementar un broadcast fiable, seleccionamos como protocolo de transporte TCP, y sobre éste realizaremos las comprobaciones del nivel de aplicación necesarias.

- Dado que el protocolo de transporte que vamos a utilizar es TCP se deberá establecer una conexión entre el proceso que realiza el broadcast y los receptores. Por tanto, el protocolo **será orientado a conexión**. Esto implica que deberá existir un establecimiento de la conexión entre el cliente y servidor previo al intercambio de los datos. Además, el servidor será sin estado es decir, no es necesario mantener información global ni información de cada cliente sobre peticiones anteriores del servicio.

- El **protocolo de servicio**: El protocolo de servicio define el intercambio de mensajes entre el proceso cliente y el proceso servidor además el formato de los mensajes. En este caso se envía un mensaje de petición desde el proceso que realiza el broadcast al resto de procesos y un mensaje de respuesta desde el proceso receptor al proceso que ejecuta el broadcast. Vamos a considerar que el mensaje a enviar es un array de caracteres de tamaño determinado y el mensaje de respuesta es otro array de caracteres del mismo tamaño.

d.2 Implementar el código que permite conectar a todos los procesos entre sí en C.

El proceso que invoca la función broadcast deberá previamente conectarse al resto de procesos. Para ello invocará la función conectar, que se implementa como sigue:

```
/* Variables globales */
struct sockaddr_in direcciones[N];
int conexiones[N];
int mi_id=obtener_ID();          /* Asumimos que sabemos cuál es nuestro ID */
int puerto= 10000;

       int conectar(){
              struct hostent *hp;

              int i=0, ret=0;
```

```
        hp = gethostbyname("168.222.12.12");
        for(i=0;i<N;i++){
           /* se asocia la dirección al puerto y la IP */
               bzero((char *)&server_addr, sizeof(server_addr));
               direcciones[i].sin_family = AF_INET;
               memcpy (&(direcciones[i].sin_addr), hp->h_addr, hp->h_length);
               direcciones[i].sin_port   = htons(puerto+i);
        }

        for(i=0;i<N;i++){
              if (i!=mi_id){
              while ((conexiones[i] =  socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))<0) {
                    printf ("Error en el socket");
              }
              // connect
              while((ret=connect(conexiones[i],  (struct sockaddr *) &direcciones[i],
                                 sizeof(direcciones[i]))<0){
                    printf ("Error en el connect");

              }
              }/* end if*/

        }/*end for*/
  }
```

Los procesos receptores del broadcast invocarán la función "aceptar_conexion" para quedar a la espera de la conexión por parte del proceso que hace el broadcast.

```
  int aceptar_conexion(int id){
        struct sockaddr_in server_addr,  client_addr;
        int sd, sc;
        int i=0, ret=0;

        int len;
        struct hostent *hp;

        if ((sd =  socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))<0) {
              printf ("BROADCAST receiver: Error en el socket");
              exit(1);
        }
        hp = gethostbyname("168.222.12.12");

        /* se asocia la dirección al puerto */
        bzero((char *)&server_addr, sizeof(server_addr));
        server_addr.sin_family    = AF_INET;
        mempy (&(server_addr [i].sin_addr), hp->h_addr, hp->h_length);
        server_addr.sin_port      = htons(p+id);

        if (bind(sd, &server_addr, sizeof(server_addr)) < 0) {
              printf ("SERVER: Error en el bind");
              exit(1);
        }
        listen(sd, 1);

        sc = accept(sd, (struct sockaddr *)  &client_addr, &len);
        if (sc < 0){
              exit(1);
        }

        if (receive(sc, &message, size)<0){
              printf("error recepción del proceso %d", id);
              exit(1);
        }
        if (send(sc, &respuesta, size)<0){
              printf("error enviando respuesta en el proceso %d", id);
              exit(1);

        }
        close(sc);

        return 0;
```

```
}
```

d.3 Implementar el código de la función broadcast en C.

Una vez establecida la conexión sólo es necesario el envío y recepción de los mensajes y el cierre del socket.

```
int broadcast(char *message, int size){
      char respuesta[size];

      for(i=0;i<N;i++){
            if (i!=mi_id){
                  if (send(conexiones[i], message, size)<0){
                        printf("error envio al proceso %d", i);
                  }else{
                        if (receive(conexiones[i], &respuesta, size)<0){
                              printf("error recepción del proceso %d", i);
                        }else
                        printf("Recibido mensaje --%s—del proceso %d\n", respuesta, i);
                        }
                  }
            }
      }
      for(i=0;i<N;i++)
            close(conexiones[i]);

      return 0;

}
```