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 **3 horas. NO** se podrán utilizar libros, apuntes **ni** calculadoras de ningún tipo.
Alumno: ____**Grupo:** __________

Ejercicio 1 (2,5 puntos). Responda a las siguientes preguntas cortas justificando brevemente su respuesta:

a) Dado el siguiente mensaje SOAP: ¿Qué protocolo de aplicación usa? Identifique los campos principales del mensaje y describa brevemente cuál es su contenido.

Cabecera de HTTP

```
POST / engelen/calcserver.cgi HTTP/1.1
Host: websrv.cs.fsu.edu
User-Agent: gSOAP/2.7
Content-Type: text/xml; charset=utf-8
Content-Length: 464
Connection: close
SOAPAction: ""
```

Línea de petición (POST / engelen/calcserver.cgi HTTP/1.1)

Cabeceras HTTP (MIME) (Host … Connection)

Cuerpo de HTTP (SOAP Envelope)

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:c="urn:calc">
  <SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <c:add>
      <a>100</a>
      <b>200</b>
    </c:add>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Cabeceras SOAP (de <?xml …?> hasta xmlns:c="urn:calc">)

Cuerpo SOAP (de <SOAP-ENV:Body …> hasta </SOAP-ENV:Envelope>)

Este mensaje SOAP usa el protocolo de aplicación HTTP como protocolo de encapsulamiento de datos. Las partes más importantes son:

- Cabecera de HTTP: en la cabecera se distinguen los siguientes campos:
  - Línea de petición: compuesta por el método POST, seguido del servidor, seguido de la versión de HTTP.
  - Cabeceras HTTP: compuesta de una línea por cada una de las cabeceras de tipo MIME necesarias (Host, User-Agent, Content-Type, Content-Length, …)

- Cuerpo de HTTP: el cuerpo del mensaje HTTP es un mensaje SOAP (SOAP Envelope). Los mensajes SOAP se encapsulan en XML. Se componen de dos partes:
    - Cabeceras SOAP: donde se definen los mensajes de petición y respuesta a intercambiar así como sus argumentos, los tipos de datos (independientes del lenguaje), el nombre del servicio web y su localización, entre otros.
    - Cuerpo SOAP: Se incluye el nombre del servicio a invocar y sus argumentos de entrada.

Los campos anteriores se relacionan en la figura.

b) Dado el siguiente sistema distribuido compuesto por N=7 nodos. Aplique el método de votación (quórum) para:

- Determinar la combinación de nodos R (copias de lectura) y nodos W (copias de escritura) que formarían un quórum válido.

  Se debe cumplir que:
  - $R,W<N$ y
  - $R+W > N$, y
  - $W+W > N$

  Por tanto:
  1) W=4 R=4, W=4 R=5, W=4 R=6, W=4 R=7
  2) W=5 R=3, W=5 R=4, W=5 R=5, W=5 R=6, W=5 R=7
  3) W=6 R=2, W=6 R=3, W=6 R=4, W=6 R=5, W=6 R=6, W=6 R=7
  4) W=7 R=1, W=7 R=3, W=7 R=3, W=7 R=4, W=7 R=5,W=7 R=6, W=7 R=7
- Si el coste de lectura es la mitad del coste de escritura y la probabilidad de lectura p=0.35, ¿qué combinación de R y W de las anteriores sería la más eficiente? Justifique su respuesta.

  El coste del sistema es:
  Coste= p * R * costeLectura + (1-p) * W * costeEscritura

  Probabilidad de lectura=0.35
  Probabilidad de escritura=0.65
  Coste lectura=Coste escritura/2

  Dado que la operación de escritura es más probable que la operación de lectura, interesa que el sistema sea eficiente en las escrituras. Como además, la escritura cuesta el doble que la lectura, me interesa escoger la

combinación de R y W tal que W sea mínimo (escribo en menos servidores de escritura). Por tanto, la mejor combinación es W=4 y R=4.

c) Dado el siguiente sistema distribuido compuesto por N nodos, N=6, que usa votación dinámica para mantener la consistencia de las réplicas. Los nodos tienen los siguientes valores NV y SC para una determinada réplica (NV: número de versión y SC: cardinalidad de la actualización):

	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5	Nodo 6
NV	5	5	5	6	6	6
SC	6	6	6	3	3	3

- ¿Se puede actualizar la réplica en la partición {4,5,6}? Rellene apropiadamente la siguiente tabla y justifique la respuesta.

  En la partición {4,5,6} se quiere actualizar una réplica y se ejecuta el algoritmo de votación dinámica.
  N=max{NVi}=max{6,6,6}=6
  I={4,5,6}
  M=max{SCi}=max{3,3,3}=3
  si (|I|>N/2) → se puede actualizar
  Como |I|=3 y es mayor que 3/2 → se puede actualizar en {4,5,6}

	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5	Nodo 6
NV	5	5	5	7	7	7
SC	6	6	6	3	3	3

- Una fallo de red fragmenta la partición {4,5,6} en dos: {4} y{5,6}. En esta situación ¿se puede actualizar la réplica en la partición {4}?

  N=max{NVi}=max{7}=7
  I={4}
  M=max{SCi}=max{3}=3
  si (|I|>N/2) → se puede actualizar
  Como |I|=1 y es menor que 3/2 → NO se puede actualizar en {4}

d) En un cliente NFS, un programa de usuario realiza las siguientes operaciones sobre un fichero:
   1. df=open("/mnt/usuario1/foo.txt", O_RDWR)
   2. read(df,buffer,1024)
   3. replace(buffer,buffer1,100)
   4. write(df,buffer1,1024)
   5. close(df)

NOTA: Asuma que en el cliente se ejecutó la orden: mnt –t nfs /home/ /mnt

- Describir el conjunto de llamadas NFS necesarias para realizar esas operaciones y qué argumentos necesita cada una de estas llamadas.

Los servicios que proporciona el servidor NFS se ofrecen en forma de RPC. Los clientes NFS deben traducir los servicios sobre ficheros a las llamadas RPC equivalentes que son enviadas por la red al servidor NFS.

1) El servicio open se mapea a tantas llamadas a procedimiento remoto LOOKUP, como directorios remotos de búsqueda haya (más el fichero). Este procedimiento devuelve el filehandle (manejador de fichero remoto) y atributos asociados al campo de búsqueda. Dado que /mnt es un directorio local, la búsqueda del fichero comienza en el directorio 'usuario1'. Las llamadas a RPC para obtener el filehandle del fichero foo.txt son:

   lookup(rootfh, "/usuario1") devuelve (fh0, attr0)
   lookup(fh0, "/foo.txt") devuelve (fh1,attr1)

2) El servicio read se mapea a una llamada a procedimiento remoto NFSPROC3_READ

3) El servicio write se mapea a una llamada a procedimiento remoto NFSPROC3_WRITE

4) El servicio close se mapea a una llamada a procedimiento remoto NFSPROC3_COMMIT.

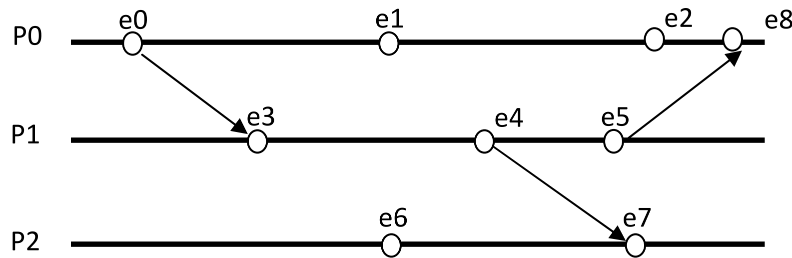
   Por tanto, son necesarias 5 RPC.

- NFS es un servidor sin estado. ¿Qué quiere decir?¿Cómo afecta esto a las datos que envía el cliente al servidor?

NFS es un servidor sin estado porque no almacena información sobre los clientes NFS. Cada invocación de los clientes al servidor de NFS es independiente, y el servidor NFS no tiene constancia de las sesiones sobre los ficheros (operaciones entre open y close), por lo que no almacena el puntero de acceso a fichero ni otras informaciones propias de la sesión. El hecho de que el servidor sea sin estado, afecta a las peticiones enviadas por los clientes en que éstas han de ser autocontenidas, es decir, deben incluir toda la información necesaria para que la operación pueda ejecutarse en el servidor. Esto lleva a que en las peticiones se envían mayor cantidad de datos.

e) ¿En qué consiste la computación voluntaria? Cite algunos ejemplos de aplicaciones.

Ejercicio 2 (1,5 puntos). Dados los siguientes procesos P1, P2 y P3 que se encuentran ejecutando en un sistema distribuido, y que producen los eventos mostrados en la siguiente figura.



Se pide:

a) Definir las relaciones de causalidad de Lamport entre los eventos que aparecen en la figura.

Lamport se basa en dos observaciones muy básicas para establecer su relación de causalidad potencial:

1. En un único procesador, se puede establecer el orden en que ejecutan los procesos.
2. El evento send(m) ocurre antes que el evento receive(m).

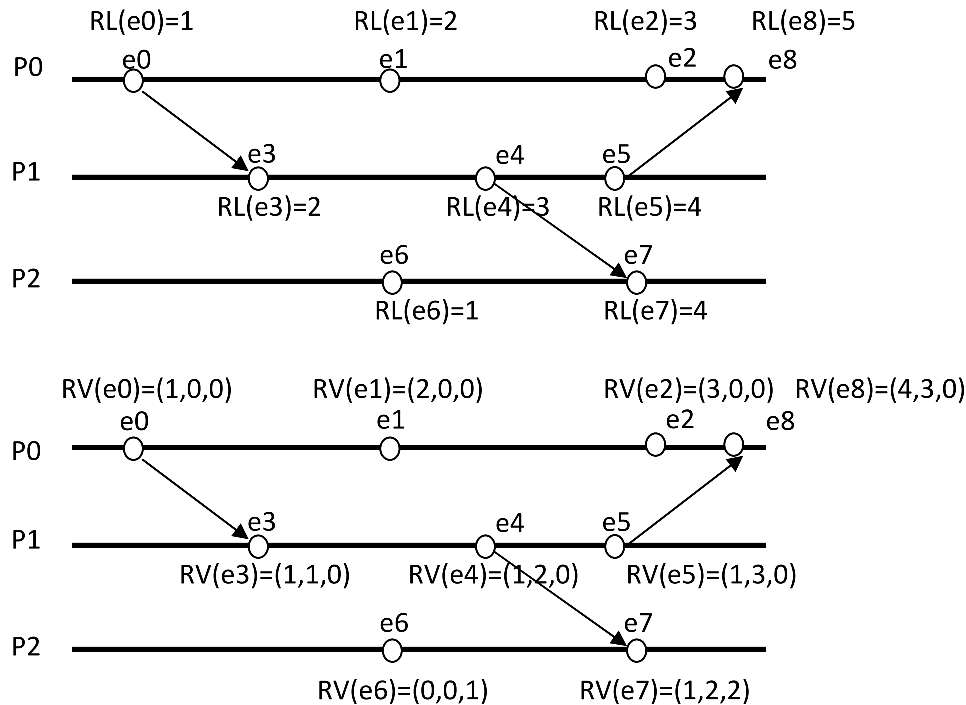
La relación de causalidad potencial de Lamport "precede a" → usa las dos observaciones anteriores para establecer un orden parcial entre eventos. En la figura, los eventos ordenados según las anteriores observaciones son:

e0→e1, e0→e2, e0→e8, e1→e2, e1→e8,e2→e8
e3→e4, e3→e5,e4→e5
e6→e7,
e0→e3,e0→e4,e0→e5,e0→e7
e3→e8,e3→e7
e4→e8

b) Definir para qué eventos no es posible establecer las relaciones de causalidad de Lamport. Justifique su respuesta.

No es posible establecer una relación de causalidad potencial entre los eventos que no satisfacen cualquiera de las dos observaciones anteriores. Estos eventos se dicen que son concurrentes y se denota como a||b. Los eventos concurrentes en el ejemplo son:
e1||e6,e1||e7,e1||e3,e1||e4,e1||e5
e2||e3,e2||e4,e2||e5,e2||e6,e2||e7
e3||e6,e4||e6,e5||e6,e5||e7
e8||e6,e8||e7
y simétricos.

c) Usando los relojes lógicos de Lamport, indique las marcas de tiempo para los eventos de los procesos anteriores.



d) Si RL1(e0)<RL2(e3), ¿sería posible decir que e0 precede a e3? Justifique su respuesta

Si RL(e0) es menor que el RL(e3) no se puede inferir que e0→e3, ni que e3→e0. Los relojes lógicos de Lamport proporcionan un orden parcial y ordenan los eventos de comunicación entre procesos y los internos en el mismo proceso. Si no se observan ninguna de estas dos cosas, los relojes no pueden ordenar los eventos y en ese caso se dicen que son concurrentes. Un contraejemplo puede observarse en los eventos e1 y e6. Aunque el RL(e1)=2 y RL(e6)=1, i.e. RL(e6)<RL(e1) no puede observarse que e6 preceda a e1.

Ejercicio 3 (2'5 puntos). Se desea implementar el algoritmo de exclusión mutua distribuida basado en coordinador usando paso de mensajes. En este algoritmo, uno de los nodos actúa como coordinador. Cuando el proceso **i** (diferente al coordinador) quiere entrar en la sección crítica envía un mensaje al coordinador:

send_entrada(i):

La función del coordinador es decidir si el proceso que solicitante puede o no entrar en la sección crítica. Si en el momento de recibir el mensaje ningún otro proceso está ejecutando la sección crítica, el coordinador permitirá al solicitante entrar y para ello le enviará el mensaje:

send_ok(i)

Si por el contrario, hay algún otro proceso ejecutando en la sección crítica, el coordinador no responderá al proceso solicitante hasta que la sección crítica quede libre. Cuando un proceso i sale de la sección crítica, enviará el mensaje al coordinador indicando que abandona la sección crítica:

send_salida (i)

Se pide:

a) Diseñe un conjunto de primitivas de recepción válidas para poder recibir los mensajes intercambiados entre un proceso i y el coordinador.
b) Implemente el pseudo-código del proceso coordinador utilizando las primitivas especificadas en este enunciado.

Solución:

a) Un conjunto de primitivas de recepción válidas deberían permitir sincronizarse con las primitivas de envío para recibir los mismos tipos de datos que se envían. Dado que en este caso, hay cierta semántica en las operaciones de envío podemos definir un conjunto de primitivas de recepción para mantener la semántica (en caso contrario no podríamos diferenciar el mensaje que se recibe). Por tanto:
    - int receive_entrada(int id), donde id es el identificador del proceso que quiere entrar en la sección crítica.
    - int receive_ok(int id), donde id es el identificador del proceso que puede entrar en la sección crítica.
    - int receive_salida(int id), donde id es el identificador del proceso que quiere salir de la sección crítica.

b) Código del coordinador.

```
SC=0;                       # SC=1 si SC ocupada y 0 en caso contrario
queue q_procesos[N];        # Cola para almacenar los procesos por orden de
                            llegada
pthread_mutex_t mutex;      # Protege el acceso a las variables compartidas
void leer_entrada(){
        while(1){
                receive_entrada(i);
                pthread_mutex_lock(&mutex);
                if (SC==0){
                        // Envio OK
                        send_ok(i);
                        SC++;
                }else{
                        // Almaceno en una cola de procesos
                        push(q_procesos, id);
                }
                pthread_mutex_unlock(&mutex);

        }

}
void leer_salida(){
        while(1){
                receive_salida(i);
                pthread_mutex_lock(&mutex);
                SC--;
                if (size(q_procesos) > 0){
                        id=pop(q_procesos);
                        send_ok(i);
                        SC++;
                }
                pthread_mutex_unlock(&mutex);
        }

}

main{
        int id_th1,id_th2;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
        pthread_create(&id_th1, &attr,leer_entrada,NULL);
        pthread_create(&id_th2, &attr leer_salida,NULL);
}
```

Ejercicio 4 (3,5 puntos). Una empresa de juegos on-line pretende implementar una versión básica del juego “Apalabrados”. En este juego, un usuario compone palabras a partir de otras palabras formadas por otros usuarios y una serie de letras aleatorias. Para que el usuario pueda empezar a jugar debe primero registrarse y posteriormente iniciar una partida con otro usuario ya registrado. Una vez iniciada la partida, los usuarios componen palabras y las envían al servidor para su validación. Si la palabra es correcta, el servidor calculará su puntuación y devolverá al usuario dicha puntuación. El objetivo del juego es obtener más puntos que el rival. La partida termina cuando decide el usuario o cuando un usuario gana la partida (se acaban todas las letras y tiene máxima puntuación).

Se desea implementar un sistema distribuido que proporcione el servicio “Apalabrados”. Los servicios básicos que se deben ofrecer son los siguientes:

1) Registro del usuario: un usuario se registra en el sistema con sus datos personales.
2) Iniciar una partida: un usuario inicia una nueva partida con otro usuario.
3) Enviar una palabra: un usuario envía una palabra en una partida empezada para poder puntuar en dicha partida.
4) Terminar una partida: un usuario decide abandonar una partida previamente iniciada.

Se pide:

a) Diseñar la aplicación cliente-servidor anterior utilizando sockets, indicando y especificando todos los aspectos necesarios para su diseño. Como parte del diseño, describa detalladamente el protocolo de servicio.

b) De acuerdo al diseño anterior, indique qué llamadas a la biblioteca de sockets utilizaría en el cliente y en el servidor y en qué orden.
NOTA: Indique al menos los argumentos más relevantes de las funciones de sockets.

c) Considerando que se emplean las RPC de Sun, defina la interfaz necesaria para poder implementar la aplicación cliente-servidor anterior.

Solución:

a) En una aplicación cliente-servidor necesitamos tener en cuenta al menos los siguientes aspectos de diseño:

Para diseñar una aplicación cliente-servidor necesitamos considerar aspectos comunes a las aplicaciones distribuidas:

- **Nombrado**: necesitamos saber cómo identificar la máquina donde ejecuta la aplicación servidora. Para ello usamos **direcciones IP estáticas** y asumiremos que el cliente conoce la IP (o el nombre) y el puerto donde ejecuta el servidor.

- **Escalabilidad**: para proporcionar un servicio escalable (que siga siendo efectivo mientras el número de peticiones aumenta) vamos a diseñar un **servidor concurrente,** es decir, aquel que puede atender simultáneamente varias peticiones de servicio al mismo tiempo. El servidor concurrente podría ser implementado con procesos convencionales o procesos ligeros, siendo ésta última la opción más eficiente. Dado que vamos a considerar un servidor concurrente necesitamos proteger aquellas variables compartidas que pudieran dar lugar a condiciones de carrera y por tanto llevar a resultados incorrectos en la aplicación (**aspectos de concurrencia y sincronización**).

- **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. Ambas aplicaciones (cliente y servidor) deben estar de acuerdo en el protocolo de transporte a utilizar.

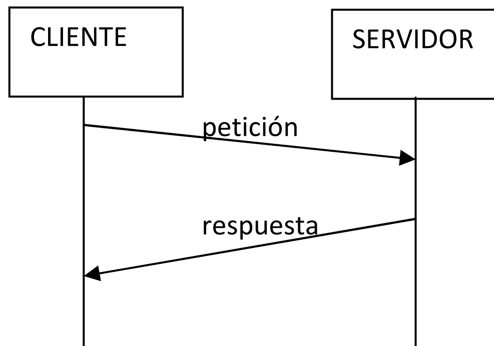
Seleccionamos **TCP** como protocolo de transporte a usar en la aplicación**.** No se necesita implementar aspectos de **QoS.** El servidor por tanto, será **orientado a conexión** dado que el protocolo de transporte que vamos a utilizar es TCP. Esto implica que deberá existir un establecimiento de la conexión entre el cliente y servidor previo al intercambio de los datos. Además, el servidor será **con estado** dado que debemos mantener **información por cliente, al menos** necesitamos saber los puntos conseguidos por cada usuario.

iv) El protocolo de servicio:

El protocolo de servicio define el intercambio de mensajes entre la aplicación cliente y servidor y además el formato de los mensajes.

Para cada servicio (registro, iniciar, terminar y enviar una palabra) vamos a definir dos mensajes: el primero de ellos lo envía el cliente al servidor, y

contiene la petición para realizar el fichaje, terminación o modificación de datos; el segundo de ellos lo envía el servidor al cliente con la respuesta.



Existen cuatro servicios que debe ofrecer el servidor:

- Registro de un usuario
- Inicio de partida
- Enviar una palabra
- Terminar una partida

Para cada petición se va a establecer una conexión, se va a enviar los argumentos de la petición y se va a recibir el resultado. Posteriormente se cierra la conexión.

Para poder identificar cada petición, se puede utilizar un byte que se envía del cliente al servidor al establecer la conexión. Se puede utilizar 0 para registro, 1 para iniciar partida, 2 para enviar una palabra y 3 terminar la partida.

Petición registro:

Se envían los siguientes datos:

- E-mail del jugador: cadena de caracteres de hasta un máximo determinado con el correo electrónico del jugador.

El mensaje de petición debe incluir además el código de petición (un 0).

El mensaje de respuesta a esta petición podría ser:

- Identificador del jugador: entero en formato big endian.

Se puede considerar que si el identificador devuelto es negativo, entonces el registro no se pudo realizar.

Iniciar una partida

Se envían los siguientes datos:

- Identificador del jugador
- El e-mail del jugador rival: cadena de caracteres de hasta un máximo determinado con el correo electrónico del jugador.

El mensaje de petición debe incluir además el código de petición (un 1).

El mensaje de respuesta a esta petición podría ser:

- Identificador de la partida: entero en formato big endian.

Se puede considerar que si el identificador devuelto es negativo, entonces la partida no se pudo iniciar.

<u>Petición enviar palabra:</u>

Se envían los siguientes datos:

- Identificador de partida: entero en formato big endian.
- Palabra: cadena de caracteres de hasta un máximo determinado con la palabra compuesta por el jugador.

El mensaje de petición debe incluir además el código de petición (un 2).

El mensaje de respuesta a esta petición podría ser:

- Número de puntos conseguidos: entero en formato big endian.

Se puede considerar que si el número de puntos devuelto es negativo, entonces la palabra enviada no es válida.

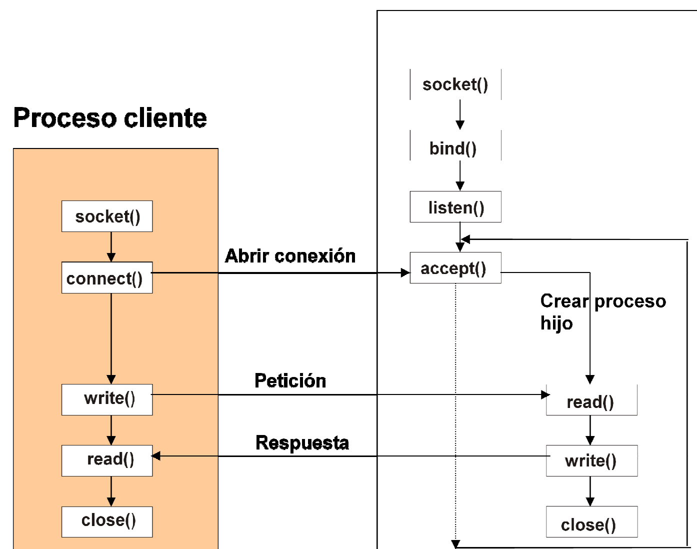
<u>Petición terminar partida:</u>

En el caso de que el jugador quiera terminar la partida, enviará:

- Identificador de partida: entero en formato big endian.

Debe incluir además el código de petición (un 3). La respuesta en este caso puede ser un byte indicando error (un -1) o éxito (un 0).

b) El modelo de comunicación TCP es el siguiente:



A continuación se detallan las llamadas a la biblioteca de sockets:

En el cliente:

1. int **socket**(int dominio, int tipo, int protocolo)
   donde dominio es AF_INET, tipo SOCK_STREAM, protocolo es 0 (elige el SO).

2. int **connect**(int socket, struct sockaddr *dir, int long)
   donde socket es el socket devuelto por socket, dir es la dirección del socket remoto y long es la longitud de la dirección.
   Esta primitiva establece la conexión con el servidor.

3. int **write**(int sd, char *buffer, int long);
   donde sd es el socket devuelto por socket, buffer es un puntero a los datos a enviar y long es el tamaño de los datos.

4. int **read(**int sd, char *buffer, int long);
   donde sd es el socket devuelto por socket, buffer es un puntero a los datos a recibir y long es el tamaño de los datos a recibir.
5. int **close**(int sd)

En el servidor:

1. int **socket**(int dominio, int tipo, int protocolo)
   donde dominio es AF_INET, tipo SOCK_STREAM, protocolo es 0 (elige el SO).

2. int **bind**(int sd, struct sockaddr *dir, int long)
   donde sd es el socket devuelto por socket, dir es la dirección del socket y long es el tamaño de la dirección.
   Habilita el socket para poder recibir conexiones entrantes.

3. int **listen**(int sd, int baklog)
   donde sd es el socket devuelto por socket, y backlog es el número de peticiones que se pueden encolar antes de que el servidor haga **accept.**

4. int **accept**(int sd, struct sockaddr *dir, int *long)
   donde sd es el socket devuelto por socket, **dir** es la dirección del cliente que ha realizado **connect** y **long** es el tamaño de la dirección.

6. int **read(**int sd, char *buffer, int long);
   donde sd es el socket devuelto por socket, buffer es un puntero a los datos a recibir y long es el tamaño de los datos a recibir.

7. int **write**(int sd, char *buffer, int long);
   donde sd es el socket devuelto por socket, buffer es un puntero a los datos a enviar y long es el tamaño de los datos.

8. int **close**(int sd)
   donde sd es el socket devuelto por socket.

c)

```
const   MAX_EMAIL=256
const   MAX_PALABRA=30

program APALABRADOS {
   version APALABRADOS1 {

   int registrar(string jugador<MAX_EMAIL>)=1;
   int partida(int jugador, string rival<MAX_EMAIL>)=2;
   int palabra(int partida, string palabra<MAX_PALABRA>)=3;
   int terminar(int partida)=4;


   }=1; /* VERSION */

} = 100000; /* PROGRAMA */
```

NOTA: No se indica como argumento el identificador de la operación dado que cuando se invoca una RPC ya se está identificando el procedimiento remoto al que se desea acceder.