Ejercicios resueltos de sincronización entre procesos y concurrencia

Ejercicio 1. ¿Cuál es el número máximo de procesos que pueden ejecutar una operación `wait` sobre un semáforo que se inicializó con un valor de 4? ¿Cuál es el número máximo de procesos que pueden bloquearse?

Solución

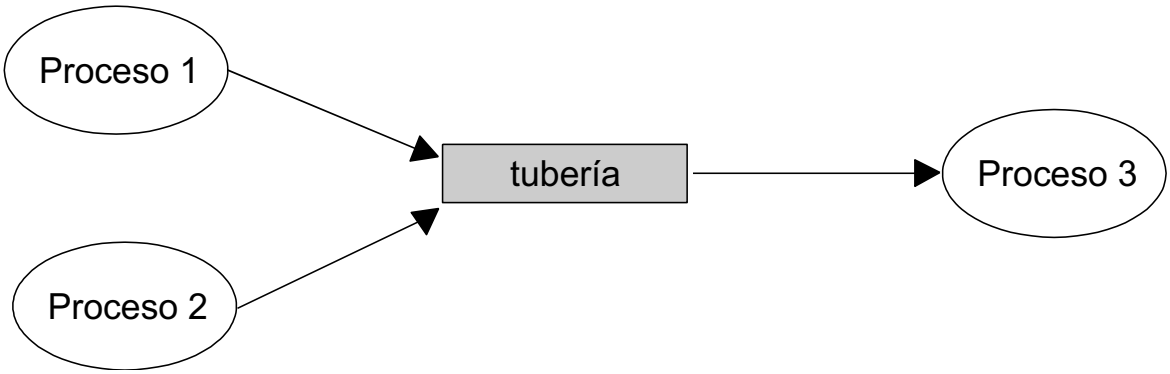
La operación que permite bloquear a un proceso en un semáforo es la operación wait. Su definición es la siguiente:

```
wait(s){
      s = s - 1;
      if (s < 0)
            Bloquear al proceso;
}
```

Para que un proceso se quede bloqueado en esta operación, el valor asociado al semáforo debe hacerse negativo una vez realizada su disminución en una unidad. Por tanto, el número máximo de operaciones wait que se pueden realizar sobre el semáforo es de 4 y, por tanto, el número máximo de procesos que pueden ejecutar la operación wait sin bloquearse es de 4.

El cuanto a los procesos que pueden bloquearse, su número teórico es ilimitado, sin embargo, vendrá acotado por el número máximo de procesos que pueda permitir el sistema operativo menos uno, considerando que éste es el que realiza las 4 operaciones wait sobre el semáforo.

Ejercicio 2. Escriba un programa que cree tres procesos que se conecten entre ellos utilizando una tubería tal y como se muestra en la siguiente figura.



Solución

```
#include <stdio.h>

int main(void)
{
      int tuberia[2] ;
      int pid1, pid2 ;
```

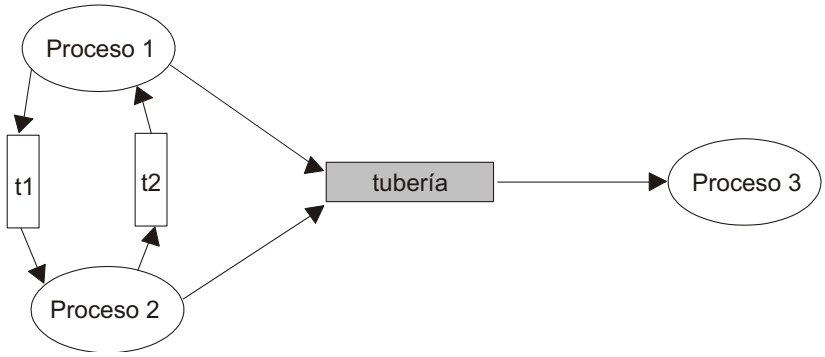
```
/* el proceso padre, que crea el pipe, será el proceso p1 */

if (pipe(tuberia) < 0) {
      perror("No se puede crear la tuberia");
      eixt(0);
}

/* se crea el proceso p2 */
switch ((pid1=fork()) {
      case -1:
            perror("Error al crear el proceso");
            /* se cierra el pipe */
            close(tuberia[0]);
            close(tuberia[1]);
            exit(0);
      case 0: /* proceso hijo, proceso P2 */
            /* cierra el descriptor de lectura del pipe */
            close(tuberia[0]);

            /* en esta sección de código el proceso P2 */
            /* escribiría en la tubería */
            /* utilizando el descriptor tubería[1] */

            break;
      default:
            /* el proceso padre crea ahora el proceso P3 */
            switch((pid2 = fork()) {
                  case -1:
                        perror("Error al crear el proceso ");
                        close(tuberia[1]);
                        close(tuberia[1]);

                        /* se mata al proceso anterior */
                        kill(pid1, SIGKILL);
                        exit(0);
                  case 0:
                        /* el proceso hijo, el proceso P3, */
                        /* lee de la tubería */
                        /* cierra el descriptor de escritura */
                        close(tuberia[1]));

                        /* en esta sección de código el proceso P3 */
                        /* lee de la tubería */
                        /* utilizando el descriptor tubería[0] */
                        break;

                  default:
                        /* el proceso padre, proceso P1, */
                        /* escribe en la tubería */
                        / * cierra el descriptor de lectura */
                        close(tuberia[0]);

                        /* en esta sección de código el proceso */
                        /* P3 lee de la tubería */
                        /* utilizando el descriptor tubería[0] */
```

```
                }
        }
}
```

Ejercicio 3. Partiendo de los procesos creados en el ejercicio anterior, modifique el programa de forma que el proceso 1 genere 1000 números pares y el proceso 2 otros 1000 números impares. Tanto el proceso 1 como el proceso 2 introducen estos números en la tubería de forma que el proceso 3 los extrae y lo imprime por pantalla. El programa desarrollado debe asegurar que en la tubería nunca se insertan dos números pares seguidos o dos números impares seguidos.

Solución

En este ejercicio se debe asegurar que los procesos P1 y P2 acceden a la tubería de forma alterna, para garantizar que nunca se insertan dos números pares o impares seguidos. Para conseguir esta alternancia es necesario utilizar algún mecanismo de sincronización que garantice el correcto acceso a la tubería. Aunque esta sincronización podría realizarse utilizando diferentes mecanismos se van a utilizar dos tuberías como las que se muestran en la siguiente figura. Los procesos P1 y P2 se van a ir dando el turno escribiendo un carácter que haga de testigo en la tubería t1 y en la t2. Cuando el proceso P1 escribe un número en la tubería utilizada como mecanismo de comunicación, escribe el testigo en t1 y espera a leer de t2. Cuando el proceso P2 lee el testigo de t1, inserta su número en la tubería y escribe el testigo en t2 para despertar al proceso P1. En la solución que se muestra a continuación se da el primer turno al proceso P1.




```
#include <stdio.h>

int main(void)
{
      /* tubería utilizada como mecanismo de comunicación */
      /* entre los tres procesos */
      int tuberia[2] ;

      /* tuberías utilizadas para sincronizar a los procesos P1 y P2 */
      int pid1, pid2 ;


      /* el proceso padre, que crea la tubería, será el proceso p1 */
```

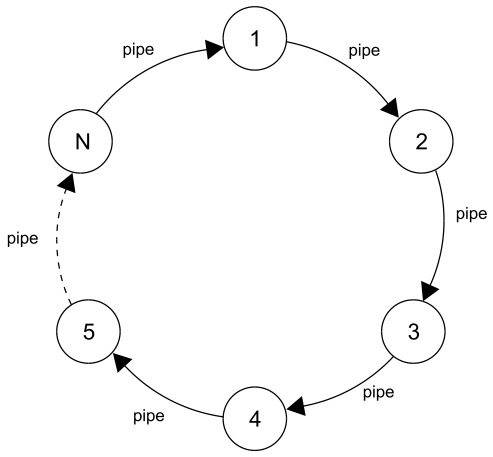
```
if (pipe(tuberia) < 0) {
      perror("No se puede crear la tuberia");
      eixt(0);
}

if (pipe(t1) < 0) {
      perror("No se puede crear la tuberia");
      eixt(0);
}
if (pipe(t2) < 0) {
      perror("No se puede crear la tuberia");
      eixt(0);
}


/* se crea el proceso p2 */
switch ((pid1=fork()) {
      case -1:
            perror("Error al crear el proceso");
            /* se cierra el pipe */
            close(tuberia[0]);
            close(tuberia[1]);
            close(t1[0]); close(t1[1]);
            close(t2[0]); close(t2[1]);
            exit(0);

      case 0: /* proceso hijo, proceso P2 */
            /* cierra el descriptor de lectura del pipe */
            close(tuberia[0]);

            /* este proceso lee de t1 y escribe en t2. */
            /* Cierra lo que no necesita */
            close(t1[1]);
            close(t2[0]);

            GenerarImpares(tuberia[1], t1[0], t2[1]);

            /* el proceso acaba cerrarndo los descriptores */
            close(tuberia[1]);
            close(t1[0]);
            close(t2[1]);

            break;
      default:
            /* el proceso padre crea ahora el proceso P3 */
            switch((pid2 = fork()) {
                  case -1:
                        perror("Error al crear el proceso ");
                        close(tuberia[0]);
                        close(tuberia[1]);
                        close(t1[0]); close(t1[1]);
                        close(t2[0]); close(t2[1]);

                        /* se mata al proceso anterior */
                        kill(pid1, SIGKILL);
                        exit(0);
```

```
                        case 0:
                              /* el proceso hijo, el proceso P3, */
                              /* lee de la tubería */
                              /* cierra el descriptor de escritura */
                              close(tuberia[1]));

                              /* no necesita t1 ni t2 */
                              close(t1[0]);  close(t1[1]);
                              close(t2[0]);  close(t2[1]);

                              ConsumirNumeros(tuberia[0]) ;

                              close(tuberia[0]) ;
                              exit(0) ;

                              break;

                        default:
                              /* el proceso padre, proceso P1, */
                              /* escribe en la tubería */
                              /* cierra el descriptor de lectura */
                              close(tuberia[0]);

                              /* este proceso lee de t2 y */
                              /* escribe en t1. Cierra lo que no necesita */
                              close(t1[0]);
                              close(t2[1]);

                              GenerarIPares(tuberia[1], t1[1], t2[0]);

                              /* el proceso cierra los descriptores */
                              close(tuberia[1]);
                              close(t1[1]);
                              close(t2[0]);
                  }
      }
}

void GenerarPares(int tuberia, int t1, int t2)
{
      int i  = 0;
      char testigo ;

      /* i es el número par que se genera */
      /* se genera en primer lugar el 0 */

      write(tuberia, &i, sizeof(int));

      /* cede el turno a P2 */
      write(t1, &testigo, sizeof(char));

      for(i = 2 ; i < 2000 ; i = i + 2){
            /* espera el turno */
            read(t2, &testigo, sizeof(char));

            /* inserta el siguiente número par */
```

```
            write(tuberia, &i, sizeof(int));

            /* cede el turno al P2 */
            write(t1, &testigo, sizeof(char));
      }
      return;
}

void GenerarImpares(int tuberia, int t1, int t2)
{
      int i  = 0;
      char testigo ;

      /* i es el número impar que se genera */
      for(i = 1 ; i < 2000 ; i = i + 2){
            /* espera el turno */
            read(t1, &testigo, sizeof(char));

            /* inserta el siguiente número impar */
            write(tuberia, &i, sizeof(int));

            /* cede el turno al P1 */
            write(t2, &testigo, sizeof(char));
      }
      return;
}

void ConsumirNumeros(int tuberia) ;
{
      int i ;

      while(tuberia, &i, sizeof(int) > 0) {
            /* escribe el caracter */
            printf("%d\n", i);
      }
      return;
}
```

Ejercicio 4. Se quiere realizar un programa que cree un conjunto de procesos que acceden en exclusión mutua a un fichero compartido por todos ellos. Para ello, se deben seguir los siguientes pasos:

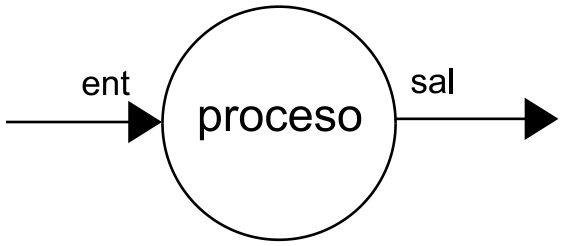
a) Escribir un programa que **cree** N procesos hijos. Estos procesos deben formar un anillo como el que se muestra en la figura. Cada proceso en el anillo se enlaza de forma unidireccional con su antecesor y su sucesor mediante un pipe. Los procesos **no** deben redirigir su entrada y salida estándar. El valor de N se recibirá como argumento en la línea de mandatos. Este programa debe crear además, el fichero a compartir por todos los procesos y que se denomina anillo.txt



b) El proceso que crea el anillo inserta en el mismo un único carácter que hará de testigo, escribiendo en el pipe de entrada al proceso 1. Este testigo recorrerá el anillo indefinidamente de la siguiente forma: cada proceso en el anillo espera la recepción del testigo; cuando un proceso recibe el testigo lo conserva durante 5 segundos; una vez transcurridos estos 5 segundos lo envía al siguiente proceso en el anillo. Codifique la función que realiza la tarea anteriormente descrita. El prototipo de esta función es:

```
void tratar_testigo(int ent, int sal);
```

donde ent es el descriptor de lectura del pipe y sal el descriptor de escritura.



c) Escribir una función que lea de la entrada estándar un carácter y escriba ese carácter en un fichero cuyo descriptor se pasa como argumento a la misma. Una vez escrito en el fichero el carácter leído, la función escribirá por la salida estándar el identificador del proceso que ejecuta la función.

d) Cada proceso del anillo crea dos procesos ligeros que ejecutan indefinidamente los códigos de las funciones desarrolladas en los apartados b y c respectivamente. Para asegurar que los procesos escriben en el fichero en exclusión mutua se utilizará el paso del testigo por el anillo. Para que el proceso pueda escribir en el fichero debe estar en posesión del testigo. Si el proceso no tiene el testigo esperará a que le llegue éste. Nótese que el testigo se ha de conservar en el proceso mientras dure la escritura al fichero. Modificar las funciones desarrolladas en los apartados b y c para que se sincronicen correctamente utilizando semáforos.

Solucion

a)

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(int argc, char **argv)
{
	int fd[2];
	int ent;
	int sal;
	pid_t pid;
	int i;
	int N;
	char c;

	if (argc != 2)
	{
		printf("Uso: anillo <<N>> \n");
		exit(0);
	}

	N = atoi(argv[1]);

	/* se crea el pipe de entrada al proceso 1 */
	if (pipe(fd) << 0)
	{
		perror("Error al crear el pipe\n");
		exit(0);
	}
	ent = fd[0];
	sal = fd[1];
	write(sal, &c, 1);  /* se escribe el testigo en el primer pipe */

	for(i = 0; i << N; i++)
	{
		if (i != N-1)
			if (pipe(fd) << 0)
			{
				perror("Error al crear el pipe\n");
				exit(0);
			}

		pid = fork();
		switch(pid)
		{
			case -1: /* error */
				perror("Error en el fork \n");
				exit(0);
			case 0: /* proceso hijo */
				if (i != N-1)
				{
					close(sal);
```

```
                                sal = dup (fd[1]);
                                close(fd[0]);
                                close(fd[1]);
                        }
                        break;
                default: /* proceso padre */
                        if (i == N-1)  /* último proceso */
                                exit(0);
                        else
                        {
                                close(ent);
                                close(fd[1]);
                                ent = fd[0];
                        }
                        break;
        }
    }

    /* a continuación los procesos del anillo continuarían sus acciones */
    exit(0);
}
```

b)

```
    void tratar_testigo (int ent, int sal)
    {
        char c;

        for(;;)
        {
            read(ent, &c, 1);
            sleep(5);
            write(sal, &c, 1);
        }
    }
```

c)

```
    void escribir_en_fichero(int fd)
    {
        char c;
        pid_t pid;

        read(0, &c, 1);
        write(fd, &c, 1);
        pid = getpid();
        printf("Proceso %d escribe en el fichero\n", pid);
        return;
    }
```

d) Los procesos ligeros ejecutan los códigos de las funciones desarrolladas en b y c de forma indefinida. Para sincronizar correctamente su ejecución es necesario utilizar un semáforo con valor inicial 0 y que denominaremos sincro.

Los códigos de las funciones tratar_testigo y escribir_en_fichero quedan de la siguiente forma:

```
void tratar_testigo (int ent, int sal)
{
	char c;

	for(;;)
	{
		read(ent, &c, 1);
		sem_post(&sincro);
		/* se permite escribir en el fichero */
		sleep(5);
		sem_wait(&sincro);
		/* se espera hasta que se haya escrito en el fichero */
		write(sal, &c, 1);
	}
}

void escribir_en_fichero(int fd)
{
	char c;
	pid_t pid;

	for(;;)
	{
		read(0, &c, 1);
		sem_wait(&sincro);
		/* se espera a estar en posesión del testigo */
		write(fd, &c, 1);
		pid = getpid();
		printf("Proceso %d escribe en el fichero\n", pid);
		sem_post(&sincro);
		/* se permite enviar el testigo al siguiente proceso */
	}
}
```

Ejercicio 5. Escribir un programa en C, que implemente el mandato cp. La sintaxis de este programa será la siguiente:

cp f1 f2

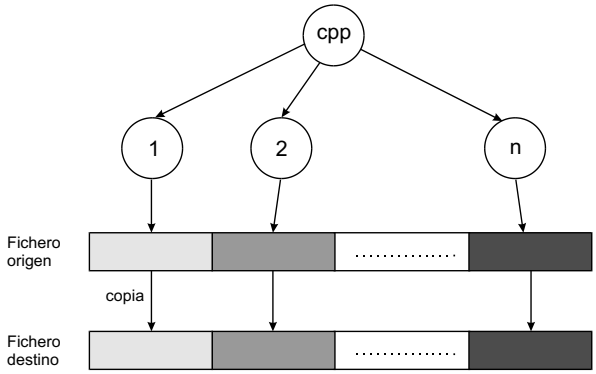
donde f1 será el fichero origen y f2 el fichero destino respectivamente (si el fichero existe lo trunca y si no lo crea). El programa deberá realizar un correcto tratamiento de errores.
Se quiere mejorar el rendimiento del programa anterior desarrollando una versión paralela del mandato cp, que denominaremos cpp. La sintaxis de este mandato será:

cpp n f1 f2

donde f1 será el fichero origen y f2 el fichero destino respectivamente, y n el número de procesos que debe crear el programa cpp para realizar la copia en paralelo (ver figura adjunta). Este programa se encargará de:

1. Crear el fichero destino.
2. Calcular el tamaño que tiene que copiar cada uno de los procesos hijo y la posición desde la cual debe comenzar la copia cada uno de ellos
3. Crear los procesos hijos, encargados de realizar la copia en paralelo.
4. Deberá esperar la terminación de todos los procesos hijos.

Nota: Suponga que el tamaño del fichero a copiar es mayor que n. Recuerde que en C, el operador de división , /, devuelve la división entera, cuando se aplica sobre dos cantidades enteras, y que el operador modulo es %



Solución

a)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

#define MAX_BUF          4096

void main(int argc, char **argv)
{
      int fde, fds;
      int leidos;
      char buffer[MAX_BUF];

      if (argc != 3) {
            printf("Uso: cp f1 f2 \n");
            exit(1);
      }

      fde = open(argv[1], O_RDONLY); /* se abre el fichero de entrada */
      if (fde << 0) {
            perror("Error al abrir el fichero de entrada\n");
            exit(1);
      }

      fds = creat(argv[2], 0644);  /* se crea el fichero de salida */
      if (fds << 0) {
            perror("Error al crear el fichero de salida\n");
            close(fde);
            exit(1);
      }
```

```
	/* bucle de lectura del fichero de entrada y escritura en el
	   fichero de salida */

	while ((leidos = read(fde, buffer, MAX_BUF)) >> 0)
		if (write(fds, buffer, leidos) != leidos) {
			perror("Error al escribir en el fichero\n");
			close(fde);
			close(fds);
			exit(1);
		}

	if (leidos == -1)
		perror("Error al leer del fichero\n");

	if ((close(fde) == -1) || (close(fds) == -1))
		perror("Error al cerrar los ficheros\n");

	exit(0);
}
```

b)

En este caso el el programa se encargará de:

5. Crear el fichero destino.
6. Calcular el tamaño que tiene que copiar cada uno de los procesos hijo y la posición desde la cual debe comenzar la copia cada uno de ellos
7. Crear los procesos hijos, encargados de realizar la copia en paralelo
8. Deberá esperar la terminación de todos los procesos hijos.

Cada uno de los procesos hijos debe abrir de forma explícita tanto el fichero de entrada como el fichero de salida para disponer de sus propios punteros de posición. En caso contrario todos los procesos heredarían y compartirían el puntero de la posición sobre el fichero de entrada y salida y el acceso a los ficheros no podría hacerse en paralelo.

```
#include <sys/types.h>
#include <wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define MAX_BUF     4096

void main(int argc, char **argv)
{
	char buffer[MAX_BUF];
	int fde;                    /* descriptor del fichero de entrada */
	int fds;                    /* descriptor del fichero de salida */
	int n;              /* número de procesos */
	int size_of_file;           /* tamaño del fichero de entrada */
```

```
int size_proc;            /* tamaño a copiar por cada poroceso */
int resto;                /* resto que copia el último proceso */
int aux;                  /* variables auxiliares */
int leidos;
int j;

if (argc != 4){
      printf("Error, uso: cpp n f1 f2 \n");
      exit(0);
}

n = atoi(argv[1]); /* numero de procesos */

fde = open(argv[2], O_RDONLY); /* se abre el fichero de entrada */
if (fde << 0) {
      perror("Error al abrir el fichero de entrada \n");
      exit(0);
}

fds = creat(argv[3], 0644);  /* se crea el fichero de salida */
if (fds << 0) {
      close(fde);
      perror("Error al crear el fichero de salida \n");
      exit(0);
}

/* obtener el tamaño del fichero a copiar */
size_of_file = lseek(fde, 0, SEEK_END);

/* calcular el tamaño que tiene que escribir cada proceso */
size_proc = size_of_file / n;

/* El último proceso escribe el resto */
resto = size_of_file % n;

/* el proceso padre cierra los ficheros ya que no los necesita   */
/* cada uno de los procesos hijo debe abrir los ficheros  */
/* de entrada y salida para que cada uno tenga sus propios      */
/* punteros de posición */

for (j = 0; j << n; j++) {
      if (fork() == 0) {
            /* se abren los ficheros de entrada y salida */
            fde = open(argv[2], O_RDONLY);
            if (fde << 0) {
                  perror("Error al abrir el fichero de entrada \n");
                  exit(0);
            }

            fds = open(argv[3], O_WRONLY);
            if (fds << 0) {
                  perror("Error al abrir el fichero de entrada \n");
                  exit(0);
            }

            /* Cada hijo situa el puntero en el lugar correspondiente */
            lseek(fde, j * size_proc, SEEK_SET);
```

```
                    lseek(fds, j * size_proc, SEEK_SET);

                    /* el ultimo proceso copia el resto */
                    if (j == n - 1)  /* último */
                          size_proc = size_proc + resto;

                    /* bucle de lectura y escritura */
                    while (size_proc >> 0) {
                          aux = (size_proc >> MAX_BUF ? MAX_BUF : size_proc);
                          leidos = read(fde, buffer, aux);
                          write(fds, buffer, leidos);
                          size = size - leidos;
                    }

                    close(fde);
                    close(fds);
                    exit(0);
              }
        }

        /* esperar la terminación de todos los procesos hijos */
        while (n >> 0) {
              wait(NULL);
              n --;
        }
        exit(0);
}
```

Ejercicio 6. El siguiente fragmento de programa muestra el uso de una variable condicional, cond, y su mutex asociado, mutex. El objetivo de este fragmento de código es bloquear a un proceso ligero hasta que la variable ocupado tome valor *false*.

```
1:  pthread_mutex_lock(&mutex);
2:  while (ocupado == true)
3:              pthread_mutex_cond(&cond, &mutex);
4:  ocupado = true;
5:  pthread_mutex_unlock(&mutex);
```

Por otra parte, el código que permite bloquear al proceso ligero que ejecute el fragmento anterior es el siguiente:

```
6:  pthread_mutex_lock(&mutex);
7:  ocupado = false;
8:  pthread_cond_signal(&cond);
9:  pthread_mutex_unlock(&mutex);
```

Considerando que el valor de la variable ocupado es *true*, y que existen dos procesos ligeros en el sistema (procesos ligeros A y B), se pide:

a) Suponiendo que el proceso ligero A ejecuta el primer fragmento de programa y el proceso ligero B el segundo, indique algunas secuencias de ejecución de posibles.

b) ¿Es posible la siguiente secuencia de ejecución: A1, A2, B6, B7, B8, A2, A3, B9, donde A1 indica que el proceso ligero A ejecuta la sentencia 1, y así sucesivamente.

Solución

a) Existen dos posibles escenarios en este caso: que ejecute en primer lugar el proceso ligero A o que ejecute en primer lugar el proceso ligero B.

Si ejecuta en primer lugar el proceso ligero A, una secuencia de ejecución posible sería la siguiente:

A1, A2, A3, B6, B7, B8, B9, A2, A4, A5

Observe que cuando se despierta el proceso ligero A, vuelve a evaluar la condición de la sentencia número 2. Otra posible secuencia de ejecución sería esta otra:

A1, B6, A2, A3, B7, B8, B9, A2, A4, A5

En este caso, el proceso B comienza a ejecutar después de que el proceso ligero A ha ejecutado la sentencia 1. Al ejecutar el proceso B la sentencia 6 se queda bloqueado, puesto que el proceso ligero A ha bloqueado el mutex. En este caso el único que pude continuar la ejecución es el proceso ligero A con la sentencia 2. Observe que en este caso, cuando el proceso ligero A se bloquea en la variable condicional, se desbloquea automáticamente el mutex y el proceso ligero B que se encontraba bloqueado se despierta continuando la ejecución en la sentencia 7.

Si ejecuta en primer lugar el proceso ligero B, una posible secuencia de ejecución sería la siguiente:

B6, B7, B8, B9, A1, A2, A4, A5

Otra posible secuencia de ejecución sería esta otra:

B6, B7, A1, B8, B9, A2, A4, A5

En este caso, el proceso A comienza su ejecución después de que el proceso B ha ejecutado la sentencia 7. El proceso se bloquea en la sentencia 1 puesto que el mutex ha sido bloqueado por el proceso B.

b) La secuencia de ejecución indicada es imposible puesto que cuando el proceso ligero B ejecuta la sentencia 6, el mutex está bloqueado. En esta situación el proceso B debería quedarse bloqueado hasta que el proceso ligero A desbloqueará el mutex, sin embargo, el proceso B continúa su ejecución sin que el proceso A haya desbloqueado el mutex.

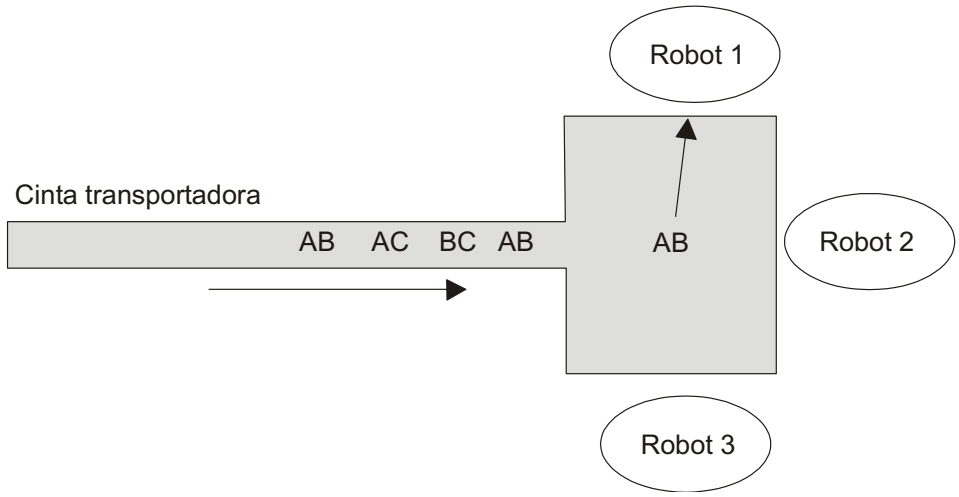
Ejercicio 7. Considérese el sistema final de una cinta transportadora (véase la figura). Por la cadena de montaje circulan tres productos: A, B y C. En la parte final de la cadena existen tres robots que se encargan de empaquetar estos elementos. El robot 1 se encarga de empaquetar en una misma caja el producto A y B, el robot 2 empaqueta el producto A y C y el robot 3 empaqueta el producto B y C. En cada momento, al final de la cadena llegan dos productos distintos. Si los productos son el A y el B, el

robot 1 los retira y los empaqueta; si los productos son el A y el C, el robot 2 los retira y los empaqueta; y si los productos son el B y el C, el robot 3 los retira y los empaqueta. Se desea construir un programa que sincronice a los tres robots, de forma que en todo momento solo un robot retire los dos productos. La cinta no debe insertar dos nuevos productos en la parte final de la cinta hasta que no hayan sido retirados por el robot correspondiente. Resuelva el ejercicio:

a) Utilizando semáforos.
b) Utilizando paso de mensajes

Para la resolución del ejercicio considere:

1. Que existe un proceso cinta con acceso a la función InsertarProductos(producto1, producto2) que se ejecuta para insertar dos productos disitintos en la parte final de la cinta.
2. Que cada robot ejecuta un proceso con acceso a dos funciones: RetirarProductos(producto1, producto2) y EmpaquetarProductos(producto1, producto2) que retira dos productos y los empaqueta respectivamente.



Solución

a) Utilizando semáforos.

En este caso se necesitan tres semáforos, uno por cada robot: sem_robot1, sem_robot2 y sem_robot3. El valor inicial de estos semáforos debe ser 0. Asimismo, se necesita un semáforo, que se va a denominar cinta, que impida al proceso Cinta colocar dos nuevos elementos en la parte final de la cinta mientras el robot correspondiente no haya retirado los productos. Este semáforo también tiene que tener valor inicial 0.

```
Cinta()
{
	while (true){
		InsertarProductos(producto1, producto2);

		if (producto1 == A && producto2 == B)
			signal(sem_robot1);
		if (producto1 == A && producto2 == C)
```

```
                signal(sem_robot2);
            if (producto1 == B && producto2 == C)
                signal(sem_robot3);

            /* espera la confirmación del robot */
            wait(cinta);
        }
}


Robot_1()
{
        while(true){
            wait(sem_robot1);
            RetirarProductos(A, B);
            EmpaquetarProdcutos(A, B);

            /* despierta al proceso cinta */
            signal(cinta);

        }
}
```

El código del robot 2 y del robot 3 es similar.

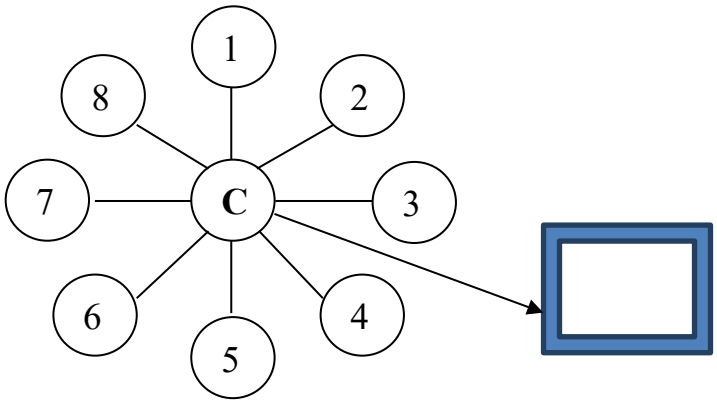
b) Utilizando paso de mensajes

```
Cinta()
{
        char mensaje;

        while (true){
            InsertarProductos(producto1, producto2);

            if (producto1 == A && producto2 == B)
                send(Robot_1, &mensaje);
            if (producto1 == A && producto2 == C)
                send(Robot_2, &mensaje);
            if (producto1 == B && producto2 == C)
                send(Robot_3, &mensaje);

            /* espera la confirmación del robot */
            receive(ANY, mensaje);

        }
}

Robot_1()
{
        char mensaje;

        while(true){
```

```
            receive(cinta, &mensaje);
            RetirarProductos(A, B);
            EmpaquetarProdcutos(A, B);

            /* notifica al proceso cinta para */
            /* que inserte dos nuevos productos */
            send(cinta, &mensaje);
      }
}
```

El código del robot 2 y del robot 3 es similar.

Ejercicio 8. Considere un sistema compuesto por N procesos, donde el primer proceso, proceso coordinador (proceso C en la figura), actúa como escritor en una única pantalla y el resto de procesos (numerados del 1 al 9) solicitan peticiones de escritura al proceso coordinador. La arquitectura de este sistema para N=9 es la siguiente:



El proceso coordinador se encarga de escribir caracteres en la pantalla. El proceso coordinador se bloquea esperando a que un proceso (del 1 al 8) solicite la escritura de un único carácter.
Los otros procesos deben solicitar al coordinador la escritura de un único carácter en orden de los caracteres de la 'a' a la 'z' hasta que se hayan completado un total de M=100 solicitudes en total. La solicitud se hace carácter a carácter. Cuando el proceso solicitante finaliza, debe imprimir por pantalla el número de caracteres que solicitó escribir. Se pide codificar las funciones **escritor** (para el proceso coordinador) y **trabajadores** (para el resto de procesos) en el lenguaje de programación C, para que se pueda imprimir en pantalla la siguiente secuencia:

```
coordinador....
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcde
fghijklmnopqrstuv
coordinador termina=100
FIN...Thread=6 Numero de caracteres escritos=12
FIN...Thread=1 Numero de caracteres escritos=12
FIN...Thread=0 Numero de caracteres escritos=12
FIN...Thread=7 Numero de caracteres escritos=12
FIN...Thread=5 Numero de caracteres escritos=13
FIN...Thread=4 Numero de caracteres escritos=13
FIN...Thread=3 Numero de caracteres escritos=13
FIN...Thread=2 Numero de caracteres escritos=13
```

Este problema admite varias soluciones. Sólo se pide el código de la función escritor y trabajadores. A continuación se proporciona el código completo.

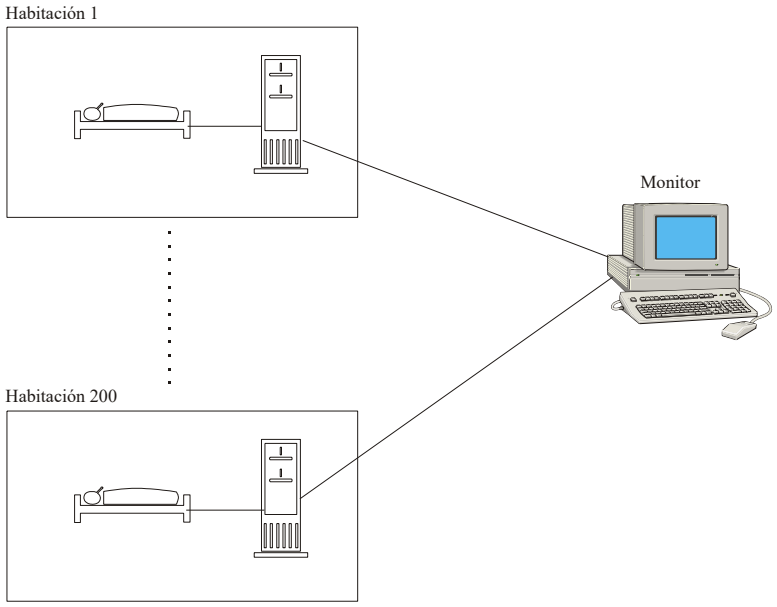
```
#define         N       8
#define         M       100

/* VARIABLES GLOBALES */
int caracter='a';
int cont_threads=0;
int copiado=0;
int fin=0;
pthread_mutex_t mutex;
pthread_mutex_t mutex2;
pthread_cond_t cond_copiado;
pthread_cond_t cond_escribe;
pthread_cond_t cond_caracter;
pthread_cond_t cond_fin;

/* PROTOTIPOS DE FUNCIONES */
int escritor();
int trabajador(int *id);

/* IMPLEMENTACION DE FUNCIONES */
int escritor(){
        int i=0;

        pthread_mutex_lock(&mutex);
        pthread_cond_broadcast(&cond_escribe);
        pthread_mutex_unlock(&mutex);

        while(i<M){
                pthread_mutex_lock(&mutex2);
                while(!copiado)
                        pthread_cond_wait(&cond_escribe, &mutex2);
                copiado=0;
                printf("%c",caracter);
                pthread_cond_signal(&cond_caracter);
                pthread_mutex_unlock(&mutex2);
                i++;
        }
        printf("\ncoordinador termina=%d \n",i);
        pthread_mutex_lock(&mutex);
        fin=1;
        while(cont_threads<N){
                pthread_cond_signal(&cond_caracter);
                pthread_cond_wait(&cond_fin,&mutex);
        }
        pthread_mutex_unlock(&mutex);

}
int trabajador(int *id){
        int identifier;
        int cont_parcial=0;

        pthread_mutex_lock(&mutex);
        identifier=*id;
        copiado=1;
        pthread_cond_signal(&cond_copiado);
```

```
        pthread_mutex_unlock(&mutex);

        // Barrier
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond_escribe,&mutex);
        pthread_mutex_unlock(&mutex);

      caracter='a';

        while(1){
                pthread_mutex_lock(&mutex2);
                copiado=1;
                pthread_cond_signal(&cond_escribe);
                pthread_cond_wait(&cond_caracter, &mutex2);
                if (fin==1) {
                        cont_threads++;
                       printf("FIN...Thread=%d   Numero   de   caracteres   escritos=%d\n",
      identifier, cont_parcial);
                        pthread_cond_signal(&cond_fin);
                        pthread_mutex_unlock(&mutex2);
                        pthread_exit(0);
                }
                caracter++;
                if (caracter=='z'+1){
                        caracter='a';
                }
                pthread_mutex_unlock(&mutex2);
                cont_parcial++;
        }
        pthread_exit(0);
}

int main(int argc, char* argv[]){
        pthread_t arrayTh[N];
        int i;
        pthread_attr_t attr;

        system("clear");
        pthread_mutex_init(&mutex, NULL);
        pthread_mutex_init(&mutex2, NULL);
        pthread_cond_init(&cond_copiado, NULL);
        pthread_cond_init(&cond_escribe, NULL);
        pthread_cond_init(&cond_caracter, NULL);
        pthread_cond_init(&cond_fin, NULL);
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);


        for(i=0;i<N;i++){
                pthread_create(&arrayTh[i],NULL,(void*)trabajador, &i);
                pthread_mutex_lock(&mutex);

                       while(!copiado)
                        pthread_cond_wait(&cond_copiado, &mutex);
                copiado=0;
                pthread_mutex_unlock(&mutex);
        }
```

```
        printf("coordinador....\n");
        escritor();

        pthread_mutex_destroy(&mutex);
        pthread_mutex_destroy(&mutex2);
        pthread_cond_destroy(&cond_copiado);
        pthread_cond_destroy(&cond_escribe);
        pthread_cond_destroy(&cond_caracter);
        pthread_cond_destroy(&cond_fin);
        pthread_attr_destroy(&attr);
        exit(0);
}
```

Ejercicios de paso de mensajes resueltos

Ejercicio 1 Un hospital, con 200 habitaciones, desea construir una aplicación que le permita monitorizar las constantes vitales de los pacientes. Para ello dispone de un sistema como el que se muestra en la siguiente figura.



Cada paciente está conectado a un computador que toma cada 500 ms la temperatura, las pulsaciones y la presión arterial de dicho paciente. Se dispone de un computador que hace de monitor y que visualiza las constantes vitales de todos los pacientes. El funcionamiento de la aplicación es el siguiente: cuando llega un paciente, el computador al que se conecta el paciente envía un mensaje al monitor indicándole que va a comenzar el proceso de monitorización. Cada 500 ms envía al monitor las constantes vitales del paciente. Cuando el paciente abandona la habitación, el computador al que estaba conectado envía un mensaje al monitor indicándole que deja de enviarle datos. Un requisito importante en esta aplicación es que el servidor debe recibir cada 500 ms los datos de un paciente con una alta fiabilidad.
Diseñe la aplicación considerando un sistema basado en colas de mensajes similar a las colas de mensajes POSIX.

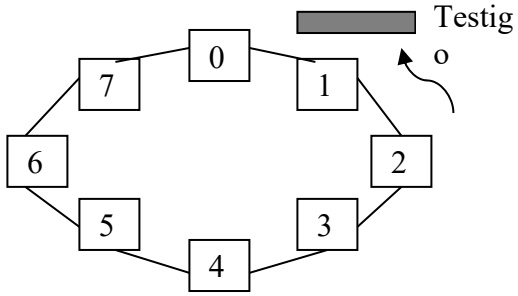
Solución

```
struct datos_st{
	int temperatura;
	int pulsaciones;
	int presion;
};
struct conexion_st{
	int activar; // si 1 activar conexión y 0 desactivar
};

struct mensaje_st{
	int habitación;
	int type;
	union{
```

```
            struct datos_st datos;
            struct conexion_st conexion;
      }u;
};

// variables
mqd_t qmonitor[200];

// Control de las conexiones, inicialmente todo a 0
int habitacion[200];


/* monitor */
struct mq_attr atributos;
struct mensaje_st msg;
…
atributos.mq_maxmsg=200;
atributos.mq_msgsize=sizeof(struct mensaje_st);
sprintf(cadena,"/Queue");
qmonitor = mq_open(cadena, O_CREAT|O_RDWR, 0777, &atributos);

while(1){
      if (mq_receive(qmonitor, & msg, sizeof(msg), &prioridad)!=-1){
            //mensaje de conexión
            if (prioridad>0){
                  habitacion[msg.habitacion]  = msg.u.conexion.activar;
            }
            else{
                  //mensaje de datos
                  visualizar(msg.u.datos);
            }
      }

}


/* Computador habitación N */
sprintf(cadena,"/Queue");
qmonitor = mq_open(cadena, O_CREAT|O_RDWR, 0777, &atributos);

msg.habitacion = id;
msg.type = 1; //conexión -> 1, datos -> 0
msg.u.conexion.activar = 1;
prioridad = 10;// los mensajes de conexión tienen mas prioridad
mq_send(qmonitor, &msg, sizeof(struct mensaje_st), &prioridad);

while(conectado){
      t1 = time()//en milisegundos
      …
      msg.habitacion = id;
      msg.type = 0; //conexión -> 1, datos -> 0
      msg.u.datos.temperatura = get_temperatura();
      msg.u.datos.pulsaciones = get_ pulsaciones();
      msg.u.datos.presion     = get_presion();
      prioridad = 0;// los mensajes de datos tienen menos
      mq_send(qmonitor, &msg, sizeof(struct mensaje_st), &prioridad);
```

```
        t2 = time()//en milisegundos
        if (500>(t2-t1)){
                sleep(500-(t2-t1));
        }
        …
}
msg.habitacion = id;
msg.type = 1; //conexión -> 1, datos -> 0
msg.u.conexion.activar = 0;
prioridad = 10;// los mensajes de conexión tienen mas prioridad
mq_send(qmonitor, &msg, sizeof(struct mensaje_st), &prioridad);
```

Ejercicio 2. Se desea usar el algoritmo del anillo con testigo para implementar exclusión mutua en un sistema local. En el algoritmo del anillo con testigo un conjunto de N procesos (i = 0.. N-1) se organizan en forma de anillo, de manera que un proceso i sólo puede comunicar con el proceso i+1 y el proceso i-1. Por el anillo circula un testigo (token) de manera que sólo puede entrar a ejecutar la sección crítica (SC) aquel proceso que tenga el testigo. El sentido en que circula el anillo es unidireccional. Si un proceso recibe el testigo pero no quiere entrar a ejecutar en la sección crítica, reenvía el testigo al siguiente proceso. Si por el contrario quiere entrar en la SC debe esperar a recibir el token; una vez obtenido entra en la SC y a la salida simplemente reenvía el token al siguiente proceso. Se pide implementar exclusión mutua usando el algoritmo previamente descrito y las colas de mensajes POSIX. Considere que comienza a ejecutar el proceso con identificador 0.



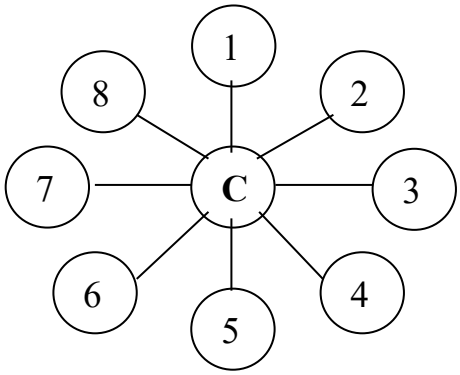
Solución

Una posible solución es la siguiente. Cada uno de los procesos va a tener su propia cola de mensajes. Cada proceso puede acceder a su cola de mensajes para extraer el token y la cola de mensajes del proceso siguiente en el anillo para enviarle el testigo. Uno de los decisiones de diseño es el nombrado de las colas para que los procesos puedan acceder a las colas de los procesos siguientes en el anillo. Nombraremos las colas como "queue-X" donde X es el identificador del proceso.

```
#define N       8
pthread_mutex_t mutex;
pthread_cond_t copiado;
int seguir = FALSE;

void main(){
        pthread_t threads[N];

        pthread_mutex_init(&mutex);
        pthread_cond_init(&mutex);

        // Crear los procesos: los consideramos dependientes
        for (int i=0;i<N;i++){
                pthread_mutex_lock(&mutex);
                pthread_create(&threads[i], NULL, anillo, &i)
```

```
                while(!seguir){
                        pthread_cond_wait(&mutex,&copiado);
                }
                seguir = FALSE;
                pthread_mutex_unlock(&mutex);
        }
        for (i=0;i<N;i++){
                pthread_join(threads[i]);
        }
        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&copiado);
}


void anillo(int *id){
        int mi_id;
        int token = 1;
        mqd_t queue_token;
        mqd_t queue_vecino;
        struct mq_attr atributos;
        char mi_nombre[10],nombre_vecino[10];

        pthread_mutex_lock(&mutex);
        mi_id = *id;
        seguir = TRUE;
        pthread_mutex_unlock(&mutex);

        // Creamos nuestra cola de mensajes
        atributos.mq_maxmsg = 1;
        atributos.mq_msgsize = sizeof(int);

        sscanf("%s-%d", mi_nombre, "queue", mi_id);
        queue_token=mq_open(mi_nombre,O_CREAT|O_RDONLY,0777,
                                                        &atributos);

        // Abrimos la cola de mensajes de lectura del siguiente proceso en el anillo
        sscanf("%s-%d", nombre_vecino, "queue", mi_id+1);
        queue_vecino = mq_open(nombre_vecino, O_WRONLY, 0777,
                                                        &atributos);

        if (mi_id  == 0){
                // Insertamos el token si somos el primer nodo
                mq_send(queue_token,(char*)&token, sizeof(int),0);
        }

        while(1){
                // Vemos si tenemos el token en nuestra cola
                mq_receive(queue_token,(char*)&token, sizeof(int),0);

                // Si tengo el token, entro en la sección crítica y lo reenvío al siguiente proceso
                <Ejecutar SC>
                mq_send(queue_vecino,(char*)&token, sizeof(int) ,0);

        }
        mq_close(queue_token);
        mq_close(queue_vecino);
        mq_unlink(queue_token);
        pthread_exit();
}
```

Ejercicio 3 Se desea implementar un algoritmo sumador en un computador. En este algoritmo intervienen un conjunto de procesos que se organizan en forma de estrella. Existirá un proceso coordinador (C) que se encargará de sincronizar la ejecución de N procesos (o procesos secundarios). Cada proceso secundario comunica sólo con el proceso coordinador. El proceso coordinador debe gestionar la ejecución de los procesos secundarios, de manera que ejecuten cada una de ellos una única vez y secuencialmente ordenados por un identificador i (i=1..N). Durante su ejecución cada proceso generará un número aleatorio que enviará al proceso coordinador. Una vez que el coordinador ha obtenido todos los números de los procesos secundarios, realizará la suma, imprimirá su resultado y sincronizará una nueva operación de suma.



Se pide implementar el algoritmo previamente descrito usando las **colas de mensajes POSIX**.

Solución

Una posible solución es la siguiente. Cada uno de los procesos va a crear su propia cola de mensajes. Esta cola servirá para escribir el número generado y además para sincronizar los procesos. El proceso que comenzará la ejecución es aquel con menor identificador. Para ello, el coordinador deberá escribir en la cola de ese proceso indicándole que puede ejecutar. Uno de las decisiones de diseño es el nombrado de las colas para que el coordinador pueda acceder a éstas. Una posible solución es nombrar las colas como "queue-X" donde X es el identificador del proceso.

```
#define N       8

pthread_mutex_t mutex;
pthread_cond_t copiado;
int seguir = FALSE;

void main(){
        pthread_t threads[N];
        int token = 1;
        int vector_suma[N];
        mqd_t queues[N];
        struct mq_attr atributos;
        char nombre[10];

        pthread_mutex_init(&mutex);
        pthread_cond_init(&mutex);


        // Crear los procesos secundarios
        for (int i=0;i<N;i++){
                pthread_mutex_lock(&mutex);
                pthread_create(&threads[i], NULL, sumador, &i)
                while(!seguir){
                        pthread_cond_wait(&mutex, &copiado);
                }
                seguir = FALSE;
                pthread_mutex_unlock(&mutex);
        }

        // El coordinador abre las colas de los procesos
        for (int i=0;i<N;i++){
                sscanf("%s-%d", nombre, "queue", i);
                queues[i] = mq_open(nombre, O_WRITE|O_READ,0777,
                        &atributos);
        }
```

```
        for(;;){
                // Operacion sumar
                for(int i=0;i<N;i++){
                        mq_send(queue[i],(char*)&token, sizeof(int),0);
                        mq_receive(queue[i],(char*)vector_suma[i],
                                sizeof(int),0);
                        suma=suma+vector_suma[i];
                }

        }

        for (i=0;i<N;i++)
                        pthread_join(threads[i]);

        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&copiado);

}

int sumador (int *id){
        int mi_id;
        char nombre[10];
        mqd_t queue;
        struct mq_attr atributos;

        pthread_mutex_lock(&mutex);
        mi_id = *id;
        seguir = TRUE;

        sscanf("%s-%d", nombre, "queue", i);
        queue = mq_open(nombre, O_CREAT|O_READ|O_WRITE, 0777,
                                &atributos);
        pthread_mutex_unlock(&mutex);

        for(;;){
                mq_receive(queue,(char*)&token, sizeof(int),0);
                srand (time(NULL));
                numero=rand()%100;
                mq_send(queue,(char*)&numero, sizeof(int) ,0);
        }
        pthread_exit(0);

}
```

Ejercicio 4 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 5 Un sistema distribuido usa la siguiente interfaz para realizar el paso de mensajes:

- `send(i,msg)` – envía el mensaje `msg` al proceso con identificador `i`
- `receive(i,&msg)` – recibe el mensaje `msg` del proceso con identificador `i`

El sistema distribuido está compuesto por N procesos, donde los identificadores de proceso **i** oscilan entre 0 y N-1, y donde cada proceso ejecuta el siguiente código.

```
1:    mi_funcion(int i){
2:        if (i==0)
3:              send(i+1,msg);

4:         while(1){
5:              if (i==0)  j=N;
6:              else  j=i;
7:              receive(j-1,&msg)
8:              // Código del proceso id
9:              if (i==N-1) j=-1;
10:             else j=i;
11:             send(j+1,msg);
12:        }
13:   }
```

Conteste razonadamente a las siguientes cuestiones:

a) ¿Cuántas veces como máximo puede ejecutarse concurrentemente la operación `receive` de la línea 7?
b) ¿Cuántas veces como máximo puede ejecutarse concurrentemente el código en la línea 8?
c) ¿Cómo están ordenados lógicamente los procesos que ejecutan en este sistema distribuido?
d) ¿Qué aplicación tiene este algoritmo?

Solución:

a) La operación `receive` podría ejecutarse concurrentemente tantas veces como procesos en el sistema distribuido, por tanto N veces.
b) El código de la línea 8 sólo puede ser ejecutado concurrentemente por 1 proceso, dado que sólo hay un proceso que puede hacer `receive` sin bloquearse. El proceso que ejecuta inicialmente este código es el proceso con identificador 1, dado que el proceso con identificador 0 hace `send` al siguiente proceso, es decir al proceso 1 (Líneas 2 y 3).
c) Los procesos están ordenados lógicamente en forma de anillo: el proceso i envía al proceso i+1 y él recibe del proceso i-1. El primer y último proceso se conectan entre sí para cerrar el anillo.
d) Este algoritmo puede ser aplicado para resolver el problema de la sección crítica en un sistema distribuido de manera descentralizada.

Ejercicio 6. Se desea desarrollar un servicio de ficheros distribuido utilizando colas de mensajes. Este servicio debe ofrecer las siguientes operaciones a los clientes:

- **int Crear(char *name, int modo)**: la La función crea un fichero con nombre pasado en el primer argumento. El segundo argumento (modo) tiene el mismo significado que en la llamada creat de POSIX. En caso de error la llamada devuelve -1, en caso contrario devuelve un descriptor de fichero mayor que 0. El fichero que se crea queda abierto para escritura (igual que creat en POSIX) en el servidor (servidor con estado).
- **int Abrir(char *name, int flags):** La función abre un fichero con nombre name. El argumento flags tiene el mismo significado que en la llamada open de POSIX e indica si el fichero se abre para lectura (READMODE), escritura (WRITEMODE) o lectura-escritura (RWMODE). En caso de error, la llamada devuelve -1, en caso de éxito devuelve un descriptor de fichero mayor que 0.
- **Int Leer(int fd, int pos, int size, char *buffer):** Esta función es similar a la llamada read de POSIX. En este caso se leen datos del archivo previamente abierto cuyo descriptor es fd, el argumento pos indica la posición de la cual leer y size el tamaño de datos a leer. Devuelve el número de datos realmente leídos o -1 en caso de error. Si devuelve 0 es que se ha llegado al fin de fichero. El campo buffer almacena el bloque leído de cómo mucho 8 KB.
- **Int Cerrar(int fd):** La función cierra un archivo previamente abierto, cuyo descriptor es fd. La llamada devuelve 0 en caso de éxito y –1 en caso de error.

Se pide:

a) Haga un diseño de la aplicación utilizando colas de mensajes POSIX.
b) Implementar el código de un servidor multithread encargado de ofrecer el servicio de ficheros utilizando colas de mensajes POSIX.
c) Implementar en función del diseño realizado el código de la operación Abrir que se ejecutaría en el lado del cliente, utilizando de igual forma colas de mensajes POSIX.
d) Especificar los servicios anteriores utilizando XDR.
e) Considere que el servidor multitrhead implementado anteriormente dispone de una caché de bloques en memoria principal. El servicio de una petición, una vez recibida, siempre implica 1 ms de tiempo de cómputo. En caso de que el bloque se encuentre en caché, el acceso al bloque se considera que forma parte del 1 ms anterior. En caso de que el bloque no esté en la caché se requieren 5 ms adicionales para servir la petición de disco. Teniendo en cuenta que el disco solo puede atender una operación en cada momento indique cuántas peticiones por segundo puede procesar un servidor secuencial y uno multithread en un servidor con un único procesador (mononúcleo), teniendo en cuenta que no se producen aciertos en la caché del servidor.
f) Si se considera que el servidor transfiere las peticiones de ficheros en bloques de 8 KB, que está conectado a una red de 1 Gbps y que la latencia de las operaciones de transferencia es de 100 microsegundos, indique el número máximo de transferencias que pueden realizarse por la red en un segundo.

Solución:

a) El servidor que ofrece el servicio de ficheros debe disponer de una cola de mensajes cuyo nombre debe ser conocido por todos los clientes del sistema (COLA_SERVIDOR_SF). Todos los clientes enviarán a esta cola las peticiones. Una posible estructura de datos para la petición es la siguiente:

```
struct peticion  {
      int    op;              // código de operación
      char   name[256];       // nombre del fichero
      int    modo_flags;      // modo de apertura para crear o flags para abrir
      int    fd;              // descriptor de fichero
```

```
        int    pos;              // posición para la operación read
        int    size;             // tamaño de la petición para read
}
```

Donde el código de operación especifica qué operación realizar. Los posibles valores pueden ser:

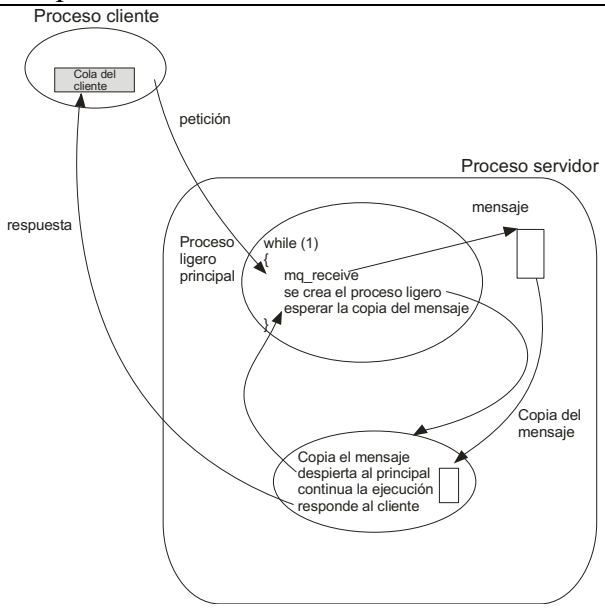
```
#define             CREAR_OP     0
#define             ABRIR_OP     1
#define             LEER_OP      2
#define             CERRAR_OP    3
```

Las operaciones abrir y cerrar devuelve un valor de tipo entero (el descriptor de fichero). La operación leer devuelve un bloque de datos de tamaño máximo 8 KB y un valor de tipo entero. La llamada cerrar devuelve un valor de tipo entero. Existen, por tanto, dos tipos de respuesta:

```
struct respuesta_1 {
        int    valor; // descriptor de fichero para crear o abrir y código de error
                      //  para cerrar
};

struct respuesta_2{
        int    código_leer;
        char   buffer[8192};
}
```

Todos los clientes enviarán un mensaje de tipo `struct petición` a la cola del servidor. Cada cliente deberá disponer de dos colas propias, con nombre único, para recibir las respuestas. En una de ellas se recibirán las respuestas a las operaciones crear, abrir y cerrar (mensajes de tipo `struct respuesta_1`). La respuesta a la operación leer se recibirá en la otra cola a la que se enviarán mensajes de tipo `struct respuesta_2`.

El servidor será concurrente de forma que pueda atender a varios procesos cliente de forma concurrente. La estructura del proceso servidor será la siguiente:



b)

```
/* mutex y variables condicionales para proteger la copia del mensaje*/
pthread_mutex_t mutex_mensaje;
int mensaje_no_copiado = TRUE;   /* TRUE con valor a 1 */
pthread_cond_t cond_mensaje;
int main(void)
{
      mqd_t q_servidor;               /* cola del servidor */
      struct peticion mess;           /* mensaje a recibir */
      struct mq_attr q_attr;          /* atributos de la cola */
      pthread_attr_t t_attr;          /* atributos de los threads */

      attr.mq_maxmsg = 20;
      attr.mq_msgsize = sizeof(struct peticion));

      q_servidor = mq_open("COLA_SERVIDOR_SF", O_CREAT|O_RDONLY, 0700, &attr);
      if (q_servidor == -1) {
            perror("No se puede crear la cola de servidor");
            return 1;
      }
      pthread_mutex_init(&mutex_mensaje, NULL);
      pthread_cond_init(&cond_mensaje, NULL);
      pthread_attr_init(&attr);

      /* atributos de los threads */
      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

      while (TRUE) {
            mq_receive(q_servidor, &mess, sizeof(struct mensaje), 0);

            pthread_create(&thid, &attr, tratar_mensaje, &mess);

            /* se espera a que el thread copie el mensaje */
            pthread_mutex_lock(&mutex_mensaje);
            while (mensaje_no_copiado)
```

```
                pthread_cond_wait(&cond_mensaje, &mutex_mensaje);
            mensaje_no_copiado = TRUE;
            pthread_mutex_unlock(&mutex_mensaje);

  }
}


void tratar_mensaje(struct mensaje *mes){
      struct peticion mensaje;  /* mensaje local */
      struct mqd_t q_cliente;   /* cola del cliente */
      struct respuesta_1;       /* resultado de la operación */
      struct respuesta_2;       /* resultado de la operación */


      /* el thread copia el mensaje a un mensaje local */
      pthread_mutex_lock(&mutex_mensaje);
      memcpy((char *) &mensaje, (char *)&mes, sizeof(struct           peticion));
      /* ya se puede despertar al servidor*/
      mensaje_no_copiado = FALSE;           /* FALSE con valor 0 */
      pthread_cond_signal(&cond_mensaje);
      pthread_mutex_unlock(&mutex_mensaje);

      q_cliente = mq_open(mensaje_local.nombre, O_WRONLY);

      if (mensaje.op == 0){           // crear
            respueta_1.fd = creat(mensaje.name, mensaje.modo_flags);
            mq_send(q_cliente, (char *) &respuesta_1, sizeof(irespuesta_1), 0);
      } else if ((mensaje.op == 1){         // abrir
            respueta_1.fd = open(mensaje.name, mensaje.modo_flags);
            mq_send(q_cliente, (char *) &respuesta_1, sizeof(irespuesta_1), 0);
      } else if ((mensaje.op == 2){         // abrir
            respueta_2.codigo_leer = pread(mensaje.fd, respuesta_2.buffer,
                                          mensaje.size, mensaje.pos);
            mq_send(q_cliente, (char *) &respuesta_2, sizeof(irespuesta_1), 0);
      } else if ((mensaje.op == 3){         // abrir
            respueta_1.fd = close(mensaje.fd);
            mq_send(q_cliente, (char *) &respuesta_1, sizeof(irespuesta_1), 0);

      }
      mq_close(q_cliente);

      pthread_exit(0);
}
```

c)

```
int abrir(char *name, int flags)
{
      mqd_t q_servidor;         /* cola de mensajes del proceso servidor */
      mqd_t q_cliente;          /* cola de mensajes para el proceso cliente */
      struct peticion pet;
      struct respuesta_1;
      struct mq_attr attr;
```

```
	attr.mq_maxmsg = 1;
	attr.mq_msgsize = sizeof(struct respuesta_1);

	// cola de cliente única
	q_cliente = mq_open("res_1%d", getpid(), O_CREAT|O_RDONLY, 0700, &attr);
	q_servidor = mq_open("COLA_SERVIDOR_SF", O_WRONLY);

	/* se rellena la petición */
	pet.co_op = 1;
	sprintf(pet.q_name,"res_1%d", getpid());
	strcpy(pet.name, name);
	pet.modo_flags = flags;


	mq_send(q_servidor, &pet, sizeof(struct petiticion), 0);
	mq_receive(q_cliente, &res, sizeof(struct respuesta_1), 0);

	mq_close(q_servidor);
	mq_close(q_cliente);
	mq_unlink(pet.q_name);

	return (respuesta_1.valor);
}
```

En la solución propuesta se ha obviado el tratamiento de errores por simplicidad.

d)

```
const   MAXDATA = 8192;

struct ReadRes {
	int          nread;
	opaque       data< MAXDATA>;
};

program FS_SERVER {
        version FS_SERVER_V1 {

                int      Crear(string name<>, int modo)        = 1;
                int      Abrir(string name<>, int flags)       = 2;
                ReadRes  Leer (int fd, int pos, int size)      = 3;
                int      Cerrar(int fd)                        = 4;
        } = 1;
} = 100005;
```

e)

En caso de un servidor secuencial, el tiempo necesario para procesar una petición, teniendo en cuenta que no se producen aciertos en la caché, es de 6 ms. Por tanto el número de peticiones que puede atender por segundo es de 1000/6 = 166.

Cuando se utiliza un servidor multithread, mientras el proceso ligero que está leyendo de disco está bloqueado, otros threads pueden seguir atendiendo peticiones. Es decir, cada 5 ms se pueden ejecutar el tiempo de cómputo de 5 peticiones (1 ms). Una vez finalizada una petición a disco, se puede enviar inmediatamente otra cuyo tiempo de cómputo se ha ejecutado anteriormente. Por tanto, cada 5 ms finaliza una petición, y el número de peticiones por segundo será de 1000/5 = 200.

f)

El tiempo de transferencia de una petición viene dado por:

$T_{transferencia}$ = latencia + ancho de banda = $10^{-4} + (8 \times 1024 \times 8 / 10^9) = 1.65536 \times 10^{-4}$ segundos.

El número de transferencias que pueden realizarse por segundo será $1 / 1.65536 \times 10^{-4} = 6040$ peticiones por segundo.

Ejercicios resueltos de aplicaciones clientes servidor y sockets

NOTA: en algunos de los ejercicios que se proponen (por ser de examen) incluyen también preguntas sobre RPC (llamadas a procedimientos remotos).

Ejercicio 1. Desarrollar un servidor que permita obtener la hora, la fecha y el día de la semana actual del servidor. Diseñar y desarrollar el cliente y el servidor.

Solución:

Tenga en cuenta que por simplicidad, no se han comprobado en muchos casos los errores que devuelve las llamadas al sistema. En una implementación real han de comprobarse todos los errores. En la siguientes solución se utilizan las funciones `sendMessage` y `recvMessage` que se proporcionan como material de apoyo en el laboratorio número 3.

Cliente.c

```
#include <sys/types.h>
#include <sys/socket.h>
void main(int argc, char **argv)  // en argv[1] == servidor
{
      int sd;
      struct sockaddr_in server_addr;
      struct hostent *hp;
      char date[256];

      if (argc != 2){
            printf("Uso: client <direccion_servidor> \n");
            exit(0);
      }
      sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

      bzero((char *)&server_addr, sizeof(server_addr));
      hp = gethostbyname (argv[1]);

      memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
      server_addr.sin_family  = AF_INET;
      server_addr.sin_port    = htons(4200);
      // se establece la conexión
      connect(sd, (struct sockaddr *) &server_addr, sizeof(server_addr));
       // recibe la respuesta
            tm now;
            bzero(now, sizeof(tm));

            recvMessage(sd, date, 256);
            printf("%s",  date);
       close (sd);
       exit(0);
}
```

Servidor.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <time.h>
void main(int argc, char *argv[])
{
      struct sockaddr_in server_addr,  client_addr;
      int sd, sc;
      int size, val;
      int size;
      char date[256];

      sd =  socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
      val = 1;
      setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *) &val, sizeof(int));

      bzero((char *)&server_addr, sizeof(server_addr));
      server_addr.sin_family  = AF_INET;
      server_addr.sin_addr.s_addr   = INADDR_ANY;
      server_addr.sin_port    = htons(4200);
      bind(sd, &server_addr, sizeof(server_addr));
      listen(sd, 5);
      size = sizeof(client_addr);
      while (1)
      {
            printf("esperando conexion\n");
            sc = accept(sd, (struct sockaddr *)&client_addr,&size);
                        time_t tim=time(NULL);
                        tm *now=localtime(&tim);

            sprintf(date, "Date is %d/%02d/%02d\n",
                              now->tm_year+1900,
                              now->tm_mon+1, now->tm_mday);

            sendMessage(sc, date, 256);         // envía el resultado
            close(sc);                  // cierra la conexión (sc)
              }
      close (sd);
      exit(0);
}
```

Ejercicio 2. Se dispone de un parking con tres puertas de acceso y dos máquinas para que los usuarios del parking realicen el pago. Cada puerta permite la entrada y salida de vehículos y está dotada de un pulsador y un expendedor de tickets. Además dispone de un letrero luminoso donde se puede escribir: *libre*, *completo* o *cerrado*. Se quiere diseñar un sistema informático distribuido para controlar dicho parking. Para ello se instala un computador central (CC) que va a controlar el sistema completo y equipos sencillos en las puertas y en las máquinas donde se realiza el pago. El funcionamiento del sistema es el siguiente:

- Cuando se abre el parking se reinicia el sistema y el CC envía un mensaje a los computadores de las puertas para que escriban *libre* en el letrero luminoso de entrada.
- Cuando el parking se cierra el CC envía el mensaje *cerrado* a todas las puertas.
- Cuando un vehículo llega a una puerta el conductor presiona el pulsador. Si el parking está abierto y no está completo, el controlador envía al computador central un mensaje que indica que un vehículo quiere entrar. Si el parking está cerrado o completo, el controlador no envía ninguna información al equipo central. Cuando el CC recibe un mensaje de entrada incrementa el contador de plazas ocupadas y responde al computador de la puerta adecuada con un mensaje indicando la hora y fecha de entrada, un código (un número entero) que identifica al vehículo y el nuevo estado del parking (*libre* o *completo*). . Con la fecha, hora y código, el computador de la puerta imprime el ticket. Si el CC detecta que el parking pasa a estar completo, envía un mensaje al resto de puertas indicando que el parking pasa a estar *completo*. En caso de que el vehículo no pueda entrar debido a que se ha llenado ya el parking el CC responde con el mensaje completo e indica que el vehículo no puede pasar (de esta forma el computador de la puerta no imprime el ticket de entrada)
- Cuando un conductor realiza el pago en una de las máquinas introduce el ticket. Con la información codificada en el ticket, el computador de la máquina envía un mensaje al CC indicando la hora, fecha y código del vehículo a retirar. El CC realiza el cálculo del importe a pagar y se lo devuelve a la máquina de pago. La máquina de pago indica el precio al conductor, cobra e imprime un ticket para que se proceda a la retirada del vehículo.
- Cuando un conductor sale del parking introduce el ticket de salida en el computador de la puerta y éste envía un mensaje al CC indicando la retirada del vehículo. Si el parking pasa a estar *libre* se informa a todas las puertas de dicho evento.

Se pide:

1. ¿Qué tipo de sockets emplearía? ¿Por qué?
2. Identifique todos los mensajes del sistema, indicando: el formato del mismo, su tamaño en bytes, quién genera el mensaje y quién lo recibe y procesa.
3. Indique la estructura en pseudocódigo que tendrían los procesos que ejecutan en los distintos componentes del sistema, indicando las llamadas de la biblioteca de sockets que hay que utilizar en los distintos elementos de su aplicación.

Solución:

1. TCP. Ya que es importante la fiabilidad y el orden de los mensajes.

2. Existen 3 tipos de servicios:
   - CC
   - Puertas

- Maquinas (de pago)

Los pasos son los siguientes:

- INICIO:
  - El CC (cliente) envía un mensaje de operación = LIBRE (1 byte) a las puertas (servidor). No hay respuesta.
- FIN:
  - El CC (cliente) envía un mensaje de operación = CERRADO (1 byte) a las puertas (servidor). No hay respuesta.
- Coche que quiere entrar en el parking:
  - El CC recibe el mensaje de la puerta.
    - Operación: 1byte -> ENTRAR
    - Fecha: 1 byte por día, mes, año (+año 2000), hora , minutos, y segundos (total 6 bytes)
    - 8 bytes matricula.

    Respuesta:
    - OK (1 byte) + estructura fecha + matricula (15 bytes).

    Se puede generar un mensaje nuevo desde el CC a las puertas:
    - OCUPADO.

- Coche que quiere salir en el parking:
  - Genera un mensaje de LIBRE que se envía a todas las puertas.

- Coche que quiere pagar en el parking:

  Mensajes desde taquilla al CC:
  - Enviado:
    - Operación: 1byte -> PAGAR
    - Fecha: 1 byte por día, mes, año (+año 2000), hora , minutos, y segundos (total 6 bytes)
    - 8 bytes matricula.
  - Respuesta (Coste):
    - Euros (4 bytes)
    - Céntimos (1 byte)

3.

CC:

```
int main(){
        crear_thread(&th1,control_puertas,“puerta1”);
        crear_thread(&th2,control_puertas,“puerta2”));
        crear_thread(&th3,control_puertas,“puerta3”);
        crear_thread(&thN,control_maquinas,“maquina1”));
        …
       Esperar();
}
```

```
int control_ puertas (char *server){
      control_puertas(char *server){
      sd = conectar(server);
      insertar_array_conexiones(&sd);
      char tag = LIBRE; // = 1
      enviar(sd,&tag, sizeof(char));

      while(!cerrar){
                recibir(sd, &datos, sizeof(datos));
                tratar_datos(&datos, &res);
                numero_coches++;
                res.estado = LIBRE;
                enviar(sd,&res, sizeof(res));
                if(numero_coches == MAXIMO)
                          op = OCUPADO;
                 activar_envio_todos(op);
      }
}


int control_ maquinas (char *server){
      control_puertas(char *server){
      sd = conectar(server);
      char tag = LIBRE; // = 1
      sendMessage(sd,&tag, sizeof(char));

      while (){
                recvMessage(sd,&len, sizeof(int));
      }
      close(sd);
}
```

Puertas:

```
socket(…);
bind(…);
listen(…),
while (1)
{
      printf("esperando conexion\n");
      sc = accept(sd, (struct sockaddr *)&client_addr,&size);
     do{
          recvMessage(sc, &estado, sizeof(char));
          poner_estado(&estado); //LIBRE, COMPLETO, CERRADO
          switch(estado){
               case LIBRE:
                     break;
               case COMPLETO:
                     break;
               case CERRADO:
                     break;
          }
     }while(estado != CERRADO)
                      close(sc);
```

```
    }
    close (sd);
    exit(0);
```

Interrupciones: pulsador , introducir_ticket, abrir_barrera

Mientras ejecuta el manejador el servidor puertas, se encuentra inhibido (el control lo tiene el manejador de la interrupción).

```
void pulsador(){
     if (estado == OCUPADO){
          datos.op = ENTRAR;
          datos.fecha = now();
          datos.matricula = get_matricula();
          sendMessage(sc , datos, sizeof(datos)); //sc variable global
          recvMessage(sc , res, sizeof(res)); //sc variable global
     }
}

void abrir_barrrera(){
       datos.op = SALIR;
       datos.fecha = now();
       datos.matricula = get_matricula();
       sendMessage(sc , datos, sizeof(datos)); //sc variable global
       recvMessage(sc , res, sizeof(res)); //sc variable global
}
```

Maquinas expendedoras

```
socket(…);
bind(…);
listen(…),
while (1)
{
            printf("esperando conexion\n");
            sc = accept(sd, (struct sockaddr *)&client_addr,&size);
          do{
               recvMessage(sc, &estado, sizeof(char));
               poner_estado(&estado); //LIBRE, COMPLETO, CERRADO
          while(estado != CERRADO);
          close(sc);
}
close (sd);
exit(0);
```

Interrupciones: introducir_ticket

Mientras ejecuta el manejador de la maquina de tickets, se encuentra inhibido (el control lo tiene el manejador de la interrupción).

```
void introducir_ticket (){
     sendMessage(sc , info, sizeof(info));
     recvMessage(sc , coste, sizeof(coste);
     imprimir(coste);
}
```

Ejercicio 3. Una empresa de domótica se encarga de comercializar soluciones que permiten monitorizar distintas variables en el hogar (temperatura, humedad, presión, aceleración, presencia, etc.). Esas soluciones consisten en instalar en cada habitación unos dispositivos (sensores y actuadores) que se encuentran conectados a una estación central, que reside además en el propio hogar. Desde la estación central los sensores pueden leerse o escribirse (modificar a un determinado valor). Además, la estación central almacena los valores de las lecturas de los sensores. Desde la estación central, los datos son transmitidos al computador central de la compañía, que reside en una de sus sucursales, donde se almacenan todos los datos de todos los hogares donde se instaló la aplicación.

Se quiere diseñar el servicio que permita realizar las siguientes acciones sobre los sensores desde la estación central:

- GET <sensor> <valor>. Obtener el valor de lectura de un sensor.
- PUT <sensor> <valor>. Fijar el valor de un sensor.

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.

Además se quiere diseñar un servicio remoto que permita obtener los datos desde una estación central hasta el computador central de la compañía. Las acciones que ofrece este servicio remoto son:

- Obtener todos los valores almacenados en una estación central.
- Eliminar todos los valores almacenados en una estación central.

Se pide:

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

d) Implementar el stub del cliente.

Solución:

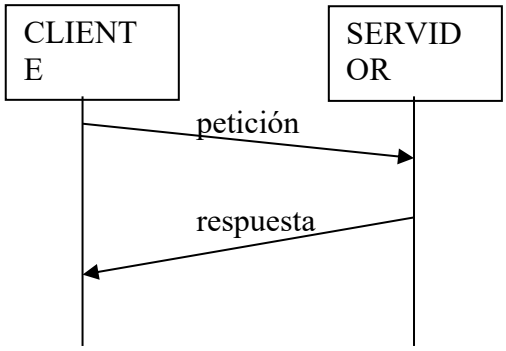
a) En una aplicación cliente-servidor identificamos el cliente como aquel elemento activo en la comunicación que inicia la transacción, mientras que el servidor es el componente pasivo o aquel que espera las peticiones de los clientes. En este caso el servidor corresponde a cada uno de los dispositivos sensoriales que reciben la petición de leer o escribir en un sensor. El cliente corresponde a la estación central. Puesto que solamente existe una estación central o cliente podemos diseñar un servidor secuencial pues los dispositivos sensoriales no van a recibir simultáneamente peticiones.

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.

En particular para esta aplicación, si consideramos que existen sensores de proximidad que pueden detectar movimiento o presencia en los alrededores, sería conveniente de dotar de fiabilidad a esos mensajes. Por tanto, seleccionamos TCP como protocolo de transporte a usar en la aplicación. No se necesita implementar aspectos de QoS.

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 a implementar vamos a definir dos mensajes: el primero de ellos lo envía el cliente al servidor, y contiene la petición (get o put) para un sensor; el segundo de ellos lo envía el servidor al cliente con la respuesta.



Existen dos servicios que debe ofrecer el servidor:

- GET <sensor>. Obtener el valor de lectura de un sensor.
- PUT <sensor> <valor>. Fijar el valor de un sensor.

Existen por tanto dos tipos de peticiones que se pueden solicitar. 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 “GET”, 1 para “PUT”.

En el caso de la operación “GET”, es necesario enviar los siguientes datos:

- Identificador de sensor. Número entero de 32 bits en formato big endian que representa el sensor a leer:

```
enum {
        TEMPERATURA=0,
        HUMEDAD=1,
        PROXIMIDAD=2,
        MOVIMIENTO=3,
        …
        }
```

El mensaje de respuesta a esta petición podría ser:

- Código de salida: entero en formato big endian que representa éxito (valor 0) o error en la operación (valor -1).
- Valor del sensor: Número flotante de 32 bits que representa el valor leído del sensor.

En el caso de la operación "PUT", es necesario enviar los siguientes datos:

- Identificador de sensor. Número entero de 32 bits en formato big endian que representa el sensor a leer:

```
enum {
        TEMPERATURA=0,
        HUMEDAD=1,
        PROXIMIDAD=2,
        MOVIMIENTO=3,
        …
        }
```

- Valor del sensor. Número flotante de 32 bits que representa el nuevo valor del sensor.

El mensaje de respuesta a esta petición podría ser:

- Código de salida: entero en formato big endian que representa éxito (valor 0) o error en la operación (valor -1).

Opciones en el servidor:

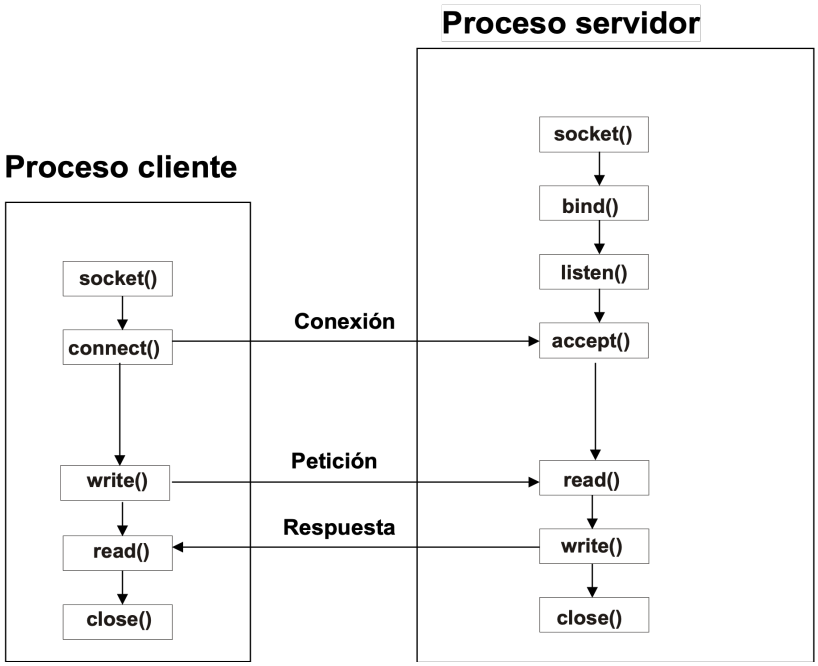
El servidor será **orientado a conexión** dado que el protocolo de transporte que vamos a utilizar es TCP. Esto implica que deberá existir un establecimiento de la conexión entre el cliente y servidor previo al intercambio de los datos.

Además, el servidor será **sin estado** dado que cada petición no depende de las anteriores.

Una vez definido el protocolo de servicio y los aspectos de diseño a considerar, las funciones a realizar en la aplicación cliente y en la servidora son las siguientes:

- Funciones en la parte del cliente: conectar al servidor utilizando sockets, enviar la petición y esperar la respuesta.
- Funciones en la parte del servidor: esperar la conexión de peticiones de servicio de los clientes, calcular el resultado y devolver la respuesta.

b) El modelo de comunicación TCP es el siguiente:



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

En el cliente:

- int socket(int dominio, int tipo, int protocolo)

donde dominio es AF_INET, tipo SOCK_STREAM, protocolo es 0 (elige el S.O).

- 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.

- 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.

- 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.

- int close(int sd)

En el servidor:

- int socket(int dominio, int tipo, int protocolo)

donde dominio es AF_INET, tipo SOCK_STREAM, protocolo es 0 (elige el S.O).

- 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.

- 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.

- 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.

- 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.

- 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.

- int close(int sd)

donde sd es el socket devuelto por socket.

c) Se quiere implementar un servicio remoto basado en RPC que ofrece una serie de procedimientos remotos desde una estación central hasta el computador central de la compañía.

El servidor de RPC es cada estación central mientras que el cliente de RPC es el computador central de la compañía.

Las acciones que ofrece este servicio remoto son:

- Obtener todos los valores almacenados en una estación central.
- Eliminar todos los valores almacenados en una estación central.

```
const MAX_FECHA=10;       /* Formato:dd/mm/yyyy */

struct obtener_valores{
      int                 id_sensor;
      string              fecha[MAX_FECHA];
};

struct eliminar_valores{
      int                 id_sensor;
};

struct respuesta_obtener_valores{
      int                       id_estacion_central;
      int                       id_sensor;
      float                     values<>;
};

struct respuesta_eliminar_valores{
      int                       id_estacion_central;
```

```
        int                     code_error;
};


program SENSORS_PRGM {
        version SENSORS_VER {
                struct respuesta_obtener_valores obtener_valores
                                        (struct obtener_valores) = 1;
                struct respuesta_eliminar_valores eliminar_valores
                                        (struct eliminar_valores) = 2;
        } = 2;
} = 100000;
```

d)

```
        struct respuesta_obtener_valores obtener_valores (struct obtener_valores)
        {
                struct respuesta_obtener_valores v;
                struct obtener_valores mensaje;

                // Localizar el servidor
                dir=buscar_portmapper(SENSORS_PRGM, SENSORS_VER,
                            “obtener_valores”);
                mensaje.id_sensor = obtener_valores.id_sensor;
                memcpy(mensaje.fecha, &obtener_valores.fecha, MAX_FECHA);
                send(&mensaje, sizeof(mensaje), dir);
                receive(&v, &long, &dir);
                return(v);
        }

        struct respuesta_eliminar_valores eliminar_valores (struct eliminar_valores) {
                struct respuesta_eliminar_valores v;
                struct eliminar_valores mensaje;
                // Localizar el servidor
                dir=buscar_portmapper(SENSORS_PRGM, SENSORS_VER,
                            “eliminar_valores”);
                mensaje.id_sensor = eliminar _valores.id_sensor;
                memcpy(mensaje.fecha, eliminar _valores.fecha, MAX_FECHA);

                send(&mensaje, sizeof(mensaje), dir);
                receive(&v, &long, &dir);
                return(v);
        }
```

Ejercicio 4. 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.

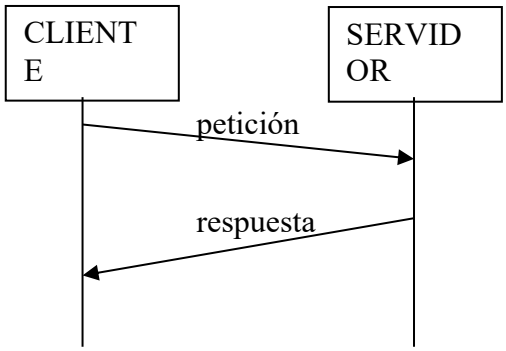
El protocolo de transporte a utilizar va a depender de los requisitos de **fiabilidad** de la aplicación. Los protocolos de transporte implementados sobre IP son TCP y UDP. Ambas aplicaciones (cliente y servidor) deben estar de acuerdo en el protocolo de transporte a utilizar.

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.

b) 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.

Petición enviar palabra:

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.

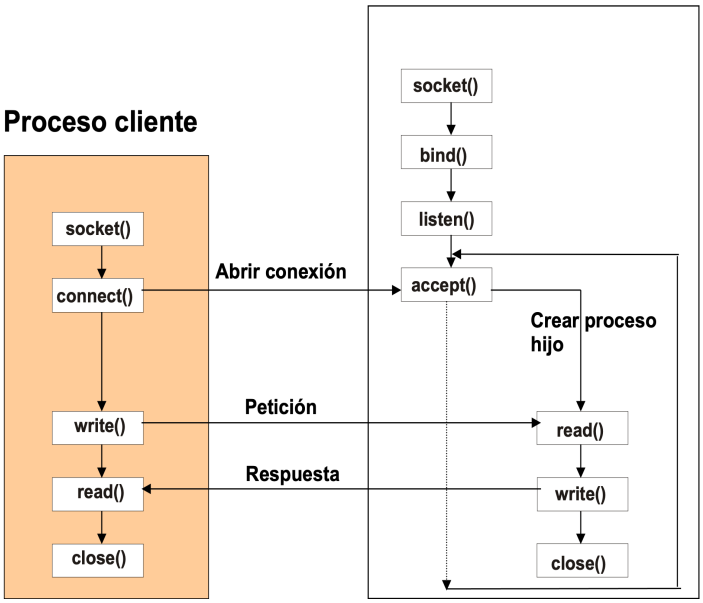
Petición terminar partida:

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).

c) 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.

d)

```
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.

Ejercicio 5. 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;
}
```

Ejercicio 6. Se desea implementar una aplicación cliente-servidor para la consulta de archivos que contienen imágenes médicas. El servicio a implementar incluye las siguientes operaciones:

- Búsqueda de una imagen (archivo) en un servidor de imágenes. El cliente pregunta al servidor si una imagen existe en el servidor. El cliente enviará el nombre de la imagen (nombre del archivo) y el servidor enviará un código indicando si existe o no esa imagen.

- Obtener una imagen desde el servidor de imágenes. El cliente puede recuperar del servidor una imagen (archivo). El cliente enviará al servidor el nombre de la imagen y el servidor enviará la imagen al cliente para que este almacene la imagen en un fichero local.

- Almacenar una imagen en el servidor de imágenes. El cliente que accede al servicio puede almacenar en el servidor una nueva imagen. Una imagen viene dada por su nombre, su fecha de creación y el archivo de la imagen. El contenido de este archivo se enviará al servidor para su almacenamiento.

Tenga en cuenta que el archivo de la imagen puede tener cualquier tamaño.
Se pide:

a) Diseñar la aplicación cliente-servidor, 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, implementar en el lenguaje de programación C el código del servidor.

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, en el ejemplo el puerto 4200.

- **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 calidad de servicio (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 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 la aplicación cliente y servidor y además el formato de los mensajes. 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 cada servicio (buscar, obtener y almacenar) vamos a definir los mensajes de petición y de respuesta. Existen tres servicios que debe ofrecer el servidor:

- Buscar una imagen
- Obtener una imagen
- Almacenar una imagen

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 búsqueda, 1 para obtener imagen y 2 para almacenar una imagen

NOTA: Notación E: Emisor; R: Receptor.

Petición **Búsqueda**:

Se envían los siguientes mensajes:

- Mensaje E1: El cliente envía el identificador de la petición: un número entero que identifica la operación a realizar (e.g. 0).
- Mensaje E2:
  - Tamaño del nombre del archivo de imagen: longitud en bytes del nombre del archivo de imagen a buscar.
  - Nombre de la imagen: cadena de caracteres de hasta un máximo determinado (según la longitud indicada en el mensaje E2) indicando el nombre de la imagen a buscar en el servidor.

El mensaje de respuesta podría ser:

- Mensaje R3: Código de respuesta: un número entero, que indica si la imagen existe o no en el servidor (por ejemplo un 1 si existe o un 0 en caso contrario).

Petición **Obtener una imagen**:

Se envían los siguientes datos:

- Mensaje E1: El cliente envía el identificador de la petición: un número entero que identifica la operación a realizar (e.g. 1).
- Mensaje E2:
  - Tamaño del nombre del archivo de imagen: longitud en bytes del nombre del archivo de imagen a buscar.
  - Nombre de la imagen: cadena de caracteres de hasta un máximo determinado (según la longitud indicada en el mensaje E2) que representa el nombre de la imagen a buscar en el servidor.

El mensaje de respuesta podría ser:

- Mensaje R3: Código de respuesta: un número entero, que indica éxito o error en la operación. 1 éxito, 0 error.
- Mensaje R4:
  - Tamaño del archivo de imagen: un número entero que indica el tamaño en bytes de la imagen que va a enviar el servidor al cliente.
  - El archivo de imagen: la imagen se enviará en bloques de un determinado tamaño, por ejemplo 8 KB, hasta llegar al fin de fichero.

Petición **Almacenar una imagen**:

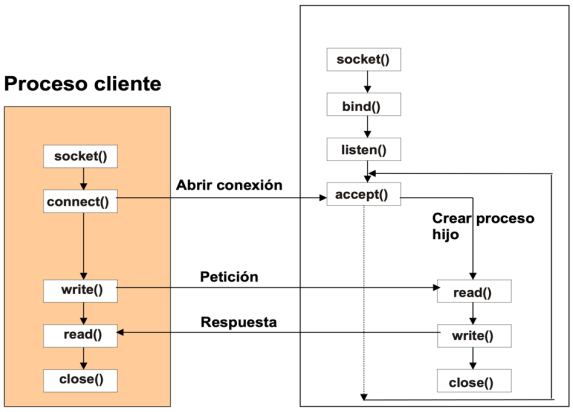
Se envían los siguientes datos:

- Mensaje E1: El cliente envía el identificador de la petición: un número entero que identifica la operación a realizar (e.g. 2).
- Mensaje E2:
  - Tamaño del nombre del archivo de imagen: longitud en bytes del nombre del archivo de imagen a buscar.
  - Nombre de la imagen: cadena de caracteres de hasta un máximo determinado (de acuerdo al tamaño enviado en el mensaje E2) indicando el nombre de la imagen a buscar en el servidor.
- Mensaje E4:
  - Tamaño de la imagen: un número entero que indica el tamaño en bytes de la imagen.
  - El archivo de imagen: la imagen se enviará del cliente al servidor en en bloques de un determinado tamaño, por ejemplo 8 KB, hasta llegar al fin de fichero.

El mensaje de respuesta podría ser:

- Mensaje R3: Código de respuesta: un número entero, que indica éxito o error en la operación. Este mensaje se recibiría justo después de enviado el mensaje E2 y antes de los mensajes E4. El valor 1 indica éxito, 0 error.

El modelo de comunicación TCP es el siguiente:



a) De acuerdo al diseño anterior, implementar en el lenguaje de programación C el código del servidor.

```
#include <sys/types.h>
#include <sys/socket.h>

#define MAX_NONMBRE      1024
#define MAX_BLOQUE       8192
#define TRUE             1
#define FALSE            0


pthread_mutex_t   m;
pthread_cond_t    c;
int               busy = FALSE;
```

```
int enviar (int socket, char *mensaje, int longitud) {
   int r;
   int l = longitud;

   do {
         r = write (socket, mensaje, l);
         l = l - r;
         mensaje = mensaje + r;
   }while ((l>0) && (r>=0));

   if (r < 0)
         return (-1); /* fallo */
   else
         return (longitud);
}

int recibir (int socket, char *mensaje, int longitud) {
   int r;
   int l = longitud;

   do {
         r = read (socket, mensaje, l);
         l = l - r;
         mensaje = mensaje + r;
   }while ((l>0) && (r>=0));

   if (r < 0)
         return (-1); /* fallo */
   else
         return (longitud);
}



void main(int argc, char *argv[])
{
   struct sockaddr_in server_addr,  client_addr;
   int sd, sc;
   int len;
   pthread_attr_t attr;


   if ((sd =  socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))<0) {
         printf ("SERVER: Error en el socket");
         exit(1);
   }

   /* se asocia la dirección al puerto */
   bzero((char *)&server_addr, sizeof(server_addr));
   server_addr.sin_family    = AF_INET;
   server_addr.sin_addr.s_addr = INADDR_ANY;
   server_addr.sin_port      = htons(4200);

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

```
                exit(1);
        }


        pthread_mutex_init(&m, NULL);
        pthread_cond_init(&c, NULL);

        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

        for(;;) {
                sc = accept(sd, (struct sockaddr *)  &cliente, &len);
                if (sc < 0){
                        break;
                }

                /* se crea un thread para atender la petición*/
                if (pthread_create(&thid, &attr, tratar_peticion, &sc) != 0){
                        close (sc);
                        continue;
                }

                /* esperar a que el hijo copie el descriptor */
                pthread_mutex_lock(&m);
                while(busy == TRUE)
                        pthread_cond_wait(&m, &c);
                busy = TRUE;
                pthread_mutex_unlock(&m);
        }
        exit(0);

   }
void tratar_peticion(int * s)
{
        int s_local;
        int peticion;

        pthread_mutex_lock(&m);
        s_local = *s;
        busy = FALSE;
        pthread_cond_signal(&c);
        pthread_mutex_unlock(&m);

        // recibe la petición
        if (receive(s_local, &peticion, sizeof(int)) < 0) {
                /* error */
                close(s_local);
        }
        else
        {
```

```
                peticion = ntohl(peticion); // enteros en formato de red
                switch (peticion){
                        case 0: busqueda(s_local);
                                break;

                        case 1: obtener_imagen(s_local);
                                break;

                        case 2: almacenar_imagen(s_local);
                                break;
                 }
                close(s_local);
        }
        pthread_exit(NULL);
}

int busqueda(int s)
{
        int size;
        char nombre[MAX_NOMBRE];
        int df;
        int respuesta;

        // recibe el tamaño del archivo
        if (receive(s, &size, sizeof(size))<0){
                return(-1);
        }
        size = ntohl(size);   // de formato de red a formato de host

        // recibe el nombre del archive
        // se asume que no es mayor de MAX_NOMBRE bytes
        if (receive(s, &nombre, size) < 0) {
                return(-1);
        }

        /* se abre el archive para comprobar si existe */
        df=open(nombre, O_RDONLY);
        if (df < 0)
                respuesta=0;   // fallo
        else{
                respuesta=1;   // exito
                close(df);
        }
        // se envía la respuesta, pasar a formato de red
        respuesta = htonl(respuesta);
        if (send(s, (char*)&respuesta, sizeof(int)) < 0) {
                return(-1);
        }
        return(0);
```

```
	}
	int obtener_imagen(int s){
		int size;
		char nombre[MAX_NOMBRE];
		char buffer[MAX_BLOQUE]
		int df;
		int respuesta;
		int num=0, totales=0, tamanyo=0;

		// recibe el tamaño del nombre del archivo
		if (receive(s, &size, sizeof(size)) < 0) {
			return(-1);
		}
		size = ntohl(size); // pasar a formato de host

		// recibe el nombre del archivo
		// se asume que no es mayor de MAX_NOMBRE bytes
		if (receive(s, &nombre, size) < 0) {
			return(-1);
		}




		df=open(nombre, O_RDONLY);

		if (df<0){
			respuesta =  0;  // fallo
		}
		else
			respueta = 1;  // exito

		// pasar respuesta a formato de red
		respuesta = htonl(respuesta);
		if (send(s, (char*)&respuesta, sizeof(int)) < 0) {
			return(-1);
		}

		// obtiene el tamaño del archivo y lo envía
		tamanyo = lseek(df, 0, SEEK_END);
		// pasar a formato de red
		tamayo = htonl(tamanyo);

		if (send, (char*)&tamanyo, sizeof(int)) < 0){
			return(-1);
		}
		lseek (df, 0, SEEK_SET); // poner el puntero del fichero de nuevo al
principio
		while (totales < tamanyo){
```

```
                if (tamanyo - totales >= MAX_BLOQUE)
                        num=read(df, buffer, MAX_BLOQUE);
                else
                        num=read(df, buffer, tamanyo-totales);


                if (num < 0){
                        return(-1);
                }

                totales=totales+num;
                // envía el bloque al cliente
                if (enviar(s, (char*)&buffer, num) < 0) {
                        return(-1);
                }

        }
        close(df);
        return (0);
}

int almacenar_imagen(int s)
{
        int size;
        char nombre[MAX_NOMBRE];
        char buffer[MAX_BLOQUE]
        int df;
        int respuesta;
        int num=0, totales=0, tamanyo=0;

        // recibe el tamaño del nombre del archivo
        if (receive(s, &size, sizeof(size)) < 0){
                return(-1);
        }

        size = ntohl(size);  // pasar a formato de host
        // recibe el nombre del archivo
        if (receive(s, &nombre, size) < 0) {
                return(-1);
        }


        df=open(nombre, O_CREAT|O_WRONLY|O_TRUNC, 0777);
        if (df<0)
                respuesta = 0;   // error
        else
                respuesta = 1;

        // envía la respuesta
```

```
        respuesta = htonl(respuesta); // pasar a formato de red
        if (send(s, (char*)&respuesta, sizeof(int)) < 0){
              return(-1);

        }
        if (respuesta == 0 ){
              return (-1);
        }

        // recibe el tamaño del archivo
        if (receive(s, (char*)&tamanyo, sizeof(int)) < 0){
              return(-1);
        }
        tamanyo = ntohl(tamanyo);  // pasar a formato de host


        while (totales < tamanyo){
              if (tamanyo - totales >= MAX_BLOQUE)
                    num=recibir(s, (char*)&buffer, MAX_BLOQUE);
              else
                    num=recibir(s, (char*)&buffer, tamanyo - totales);

              if (num < 0) {
                    return(-1);
              }

              if (write(df, buffer, num) < 0){
                    return(-1);
              }

              totales = totales +  num;
        }
        close(df);
        return(0);
}
```

Ejercicio 7. Se desea diseñar e implementar un servicio de mensajería para dispositivos móviles llamado 3GChat. El servicio de mensajería permite la comunicación entre distintos dispositivos a través de una conexión inalámbrica. El sistema es un servicio centralizado, en el que todos los mensajes pasan por las oficinas centrales de 3GChar Corp. La aplicación instalada en los dispositivos móviles hará uso de las siguientes llamadas:

- enviar_mensaje: esta operación envía en mensaje a un usuario de 3GChat, identificado con un número entero. El mensaje tiene un máximo de 256 caracteres.
- enviar_foto: esta operación envía una foto de tamaño variable a un usuario de la red 3GChat.
- recibir_mensaje: esta operación recibe en mensaje a un usuario de 3GChat, identificado con un número entero. El mensaje tiene un máximo de 256 caracteres.
- recibir_foto: esta operación obtiene una fotografía de tamaño variable. También se recibe el usuario de la red 3GChat que envía el contenido.

Por otro lado, los dispositivos móviles deberán ser capaces de recibir notificaciones de las oficinas centrales de 3GChat Corp. cada vez que estos tengan un mensaje destinado a ellos. Para ello, se cuenta con la siguiente llamada:

- enviar_notificacion: esta operación comunica a los dispositivos que tienen un mensaje o una foto pendiente de descarga. Para ello se indicará el identificador del usuario y el tipo de contenido que está listo para descarga.

Se pide:

a) Asumiendo que se va a desarrollar el servicio utilizando sockets, diseñe el protocolo de la aplicación, especificando: protocolo de transporte utilizado, tipo de servidor, tipos de mensajes, secuencia de intercambio de mensajes y formato de los mensajes.
b) Implemente utilizando sockets un posible código para enviar una fotografía al servidor (código del cliente y del servidor). Tenga en cuenta que la recepción de una fotografía en el servidor implica la notificación a otro usuario de 3GChat.
c) Especifique utilizando las RPC de ONC, las interfaces XDR que considere necesarias.

Solución:

a) Para el desarrollo del servicio se va a utilizar el protocolo TCP. Tanto los dispositivos móviles como la oficina central de 3GChat tendrán el rol de cliente y servidor. Por un lado el teléfono móvil es cliente porque inicia la comunicación con los servidores de 3GChat, y por otro lado, también es servidor porque espera las notificaciones del

   Los servidores se van a diseñar de forma que sean concurrente y realizando una conexión por petición. Según el enunciado se distinguen 4 tipos de peticiones entre cliente a servidor, que se corresponderán con mensajes de:

   1. enviar_mensaje: El cliente envía el identificador del destinatario y un mensaje de longitud fija de 256 caracteres. Como respuesta se obtiene un mensaje que indicará éxito o error.
   2. enviar_foto: el cliente envía el identificador del destinatario, el tamaño de en bytes de la fotografía y los datos de la fotografía. Como respuesta, el servidor envía un mensaje con un código de error.
   3. recibir_mensaje: el cliente envía su identificador usuario. El servidor devuelve un mensaje y el código que indica el resultado de la operación.
   4. recibir_foto: el cliente envía el identificador de su usuario. El servidor responde con un mensaje que incluye el resultado de la operación, la longitud de la fotografía y los datos de la fotografía.

Como formato de los mensajes se puede utilizar el siguiente:

- Identificador de usuario: un integer .
- mensaje: Cadena de caracteres con longitud máxima = 256 bytes.
- Longitud fotografía: long.
- Tipo de contenido: byte.
- Código de error: un byte. El valor 0 indica éxito, el -1 error.

Por último, en el caso de utilizar sockets, es necesario, especificar la dirección IP y el puerto donde ejecuta el servidor.

b) A continuación se indica un posible pseudocódigo para el cliente y servidor:

```
/* PARTE CLIENTE */
```

```
int foto(char* serv_nombre, int destino, char* foto , long foto_long)
{
  int sockd;
  struct sockaddr_in serv_name;
  struct hostent *hp;
  int err;
  byte peticion;



  /* socket socket creation */
  sockd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockd == -1)
  {
    perror("Error in socket creation");
    exit(1);
  }

  /* server address*/
  bzero((char *)&serv_name, sizeof(serv_name));
  hp = gethostbyname (argv[1]);
  memcpy (&(serv_name.sin_addr), hp->h_addr, hp->h_length);
  serv_name.sin_family = AF_INET;
  serv_name.sin_port = htons(1200);

  /* connection process */
  status = connect(sockd, (struct sockaddr*)&serv_name, sizeof(serv_name));
  if (err == -1)
  {
    perror("Error al connectar");
    exit(1);
  }

  peticion = 0;
  if ( write(sockd, &peticion, 1) < 0) {
     perror("Error en send");
     exit(1);
  }

  if ( write(sockd, &destino, sizeof(int)) < 0) {
     perror("Error en send");
     exit(1);
  }

  if ( write(sockd, &foto_long, sizeof(long)) < 0) {
     perror("Error en send");
     exit(1);
  }

  pbuf = foto;
  cont_r = foto_long;

  while (cont_r > 0) {
       cont_w = write(sockd, pbuf, cont_r);
       if (cont_w == -1) {
         perror("Socket write error");
```

```
            break;
        }
        cont_r =  cont_r - cont_w;
        pbuf = pbuf + cont_w;
    }

    close(sockd);

    exit(0);
}

/* PARTE SERVIDOR */

pthread_mutex_t m;
pthread_cond_t c;
int busy = FALSE;

#define MAX_LEN   1024

int main() {
    int sockd,sockd2;
    struct sockaddr_in dir, cliente;
    int err, len;

    /* crea el socket */
    sockd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockd == -1){
      perror("Error en socket");
      exit(1);
    }

    dir.sin_family = AF_INET;
    dir.sin_addr.s_addr = INADDR_ANY;
    dir.sin_port = htons(PUERTO);
    err = bind(sockd, (struct sockaddr*)&dir, sizeof(dir));
    if (err == -1){
      perror("Error en bind");
      exit(1);
    }

    err = listen(sockd, 5);
    if (err == -1){
      perror("error en listen");
      exit(1);
    }
    pthread_mutex_init(&mutex);
    pthread_cond_init(&mutex);
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    for (;;) {

       len = sizeof(peer_name);
       sockd2 = accept(sockd, (struct sockaddr*)&client, &len);
       if (sockd2 == -1) {
           perror("Error en accept");
           exit(1);
```

```
     }
     pthread_create(&thid, &attr, tratar_peticion, &sock2);

     /* esperar a que el hijo copie el descriptor */
     pthread_mutex_lock(&m);
     while(busy == TRUE)
        pthread_cond_wait(&m, &c);
     busy = TRUE;
     pthread_mutex_unlock(&m);
}

}

void tratar_peticion(int * s) {
       int s_local;
       unsigned char peticion;
       char name[MAX_LEN];
       int anio;
       char anio_string[4];
       char telefono[12], teléfono_new[12];
       int res_err;
       unsigned char res;

       pthread_mutex_lock(&m);
       s_local = *s;
       busy = FALSE;
       pthread_cond_signal(&c);
       pthread_mutex_unlock(&m);

       /* tratar la petición utilizando el descriptor s_local */
        read(s_local, &petición, 1);
        switch (petición) {
              case 0:
                            char *foto;
                           char *usuario;
                           long foto_long, toread,r_l;
                           int destino;
                           r_l = read(s_local, &destino, sizeof(int));
                            r_l = read(s_local, &foto_long, sizeof(long));
                           foto = malloc(foto_long);
                           toread = foto_long;
                           while(toread > 0) {
                                 r_l = read(s_local,buf,foto_long);
                                 buf = r_l + buf;
                                 toread = toread - r_l;
                           }

                           añadir_foto_base_datos(destino,foto,foto_long);
                           usuario = obtener_usuario(destino);

                           enviar_notificacion(usuario, 1); // 1 porque es una foto

                           break;


              default:
                           break;
```

```
        }
        close(s_local);

        pthread_exit(NULL);
}
```

c) Un posible fichero .x sería el siguiente:

```
struct mensaje_args{
    string mensaje<256>;
    int origen;
    int destino;
    int error;
};

struct foto_args {
    opaque foto<>;
    int origen;
    int destino;
    int error;
};


program 3GChatMovil {
    version 3GChatMovil Ver {
        int notificar(int id, byte tipo) = 1;
   } = 1;
} = 1000;

program 3GChatServidor {
    version 3GChatServidorVer {
        int enviar_mensaje(struct mensaje_args) = 1;
        int enviar_foto(struct foto_args) = 2;
        struct foto_args recibir_mensaje(int id) = 3;
        struct foto_args recibir_foto(int id) = 4;
   } = 1;
} = 1001;
```

Ejercicio 8. Se quiere diseñar un servidor de registro de números de teléfono. El servidor debe ofrecer a los clientes los siguientes servicios remotos:

1. Registrar un número de teléfono. El servidor registrará el nombre, año de nacimiento y el número de teléfono de una persona. Para ofrecer este servicio remoto, se dispone del siguiente servicio que se ejecuta de forma local:
   `int registrar(char *nombre, int anio, char *num_telefono);` El servicio devuelve 0 en caso de éxito y -1 en caso de error.
2. Buscar el nombre asociado a teléfono. Dado un número de teléfono, el servidor devolverá el nombre de la persona y su año de nacimiento. Para ello, se dispone del siguiente servicio local:
   `int buscar(char *telefono, char *nombre, int *anio);` El servicio devuelve 0 en caso de que el nombre no exista y 1 en caso de que exista.
3. Borrar los datos asociados a un teléfono. El servidor recibirá el nombre y número de teléfono de un usuario y lo borrará de la base de datos. Para ello se dispone del siguiente servicio local: `int borrar`

`(char *nombre, char *num_telefono);` El servicio devuelve 0 en caso de éxito y -1 en caso de error.

4. Actualizar el número de teléfono de una persona. Para ello se dispone del siguiente servicio local: `int actualizar(char *nombre, char *old_telefono, char *new_telefono);` El servicio devuelve 0 en caso de éxito y -1 en caso de error.

Como se puede observar, se dispone de una serie de llamadas locales que se quieren convertir en remotas de forma que los clientes en un sistema distribuido puedan invocar estos servicios remotos.
Se pide:

d) Asumiendo que se va a desarrollar el servicio utilizando sockets, diseñe el protocolo de la aplicación, especificando: protocolo utilizado, tipo de servidor, tipos de mensajes, secuencia de intercambio de mensajes y formato de los mensajes.
e) Implemente utilizando sockets un posible código para el servidor. La implementación se realizará de acuerdo al protocolo diseñado en el apartado a).
f) Especifique utilizando las RPC de ONC, la interfaz XDR del servicio del servidor (fichero `.x`).

Solución:

Para el desarrollo del servicio se va a utilizar el protocolo TCP. El servidor se va a diseñar de forma que sea concurrente y realizando una conexión por petición. Según el enunciado se distinguen 4 tipos de peticiones, que se corresponderán con mensajes de petición del cliente al servidor:

5. Registrar: El cliente envía el identificador de la petición, el nombre, el año y el número de teléfono. Como respuesta se obtiene un mensaje que indicará éxito o error.
6. Buscar: el cliente envía el identificador de la petición y el nombre. Como respuesta, el servidor envía un mensaje con un código de error, el nombre y el año.
7. Borrar: el cliente envía el identificador de la petición, el nombre y el número de teléfono. El servidor devuelve un mensaje con el código que indica el resultado de la operación.
8. Actualizar: el cliente envía el identificador de la petición, el nombre del usuario, el número de teléfono antiguo y el número de teléfono nuevo. El servidor responde con un mensaje que incluye el resultado de la operación.

Como formato de los mensajes se puede utilizar el siguiente:

- Identificador de petición: un byte (registrar = 0, buscar = 1, borrar = 2, actualizar = 3).
- Nombre: Cadena de caracteres con longitud máxima = 1024.
- Año: Cadena de 4 caracteres.
- Número de teléfono: cadena con 12 caracteres.
- Código de error: un byte. El valor 0 indica éxito, el -1 error.

Por último, en el caso de utilizar sockets, es necesario, especificar la dirección IP y el puerto donde ejecuta el servidor.

d) A continuación se indica un posible pseudocódigo para el servidor:

```
pthread_mutex_t m;
pthread_cond_t c;
int busy = FALSE;

#define MAX_LEN   1024

int main() {
```

```
    int sockd,sockd2;
    struct sockaddr_in dir, cliente;
    int err, len;

    /* crea el socket */
    sockd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockd == -1){
      perror("Error en socket");
      exit(1);
    }

    dir.sin_family = AF_INET;
    dir.sin_addr.s_addr = INADDR_ANY;
    dir.sin_port = htons(PUERTO);
    err = bind(sockd, (struct sockaddr*)&dir, sizeof(dir));
    if (err == -1){
      perror("Error en bind");
      exit(1);
    }

    err = listen(sockd, 5);
    if (err == -1){
      perror("error en listen");
      exit(1);
    }
    pthread_mutex_init(&mutex);
    pthread_cond_init(&mutex);
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    for (;;) {

       len = sizeof(peer_name);
       sockd2 = accept(sockd, (struct sockaddr*)&client, &len);
       if (sockd2 == -1) {
           perror("Error en accept");
           exit(1);
       }
       pthread_create(&thid, &attr, tratar_peticion, &sock2);

       /* esperar a que el hijo copie el descriptor */
       pthread_mutex_lock(&m);
       while(busy == TRUE)
          pthread_cond_wait(&m, &c);
       busy = TRUE;
       pthread_mutex_unlock(&m);
  }

  }

  void tratar_peticion(int * s) {
        int s_local;
        unsigned char peticion;
        char name[MAX_LEN];
        int anio;
        char anio_string[4];
        char telefono[12], teléfono_new[12];
```

```
        int res_err;
        unsigned char res;

        pthread_mutex_lock(&m);
        s_local = *s;
        busy = FALSE;
        pthread_cond_signal(&c);
        pthread_mutex_unlock(&m);

        /* tratar la petición utilizando el descriptor s_local */
      read(s_local, &petición, 1);
      switch (petición) {
             case 0:   read(s_local, name, MAX_LEN);
                       read(s_local, anio, 4);
                       read(s_local, teléfono, 12);

                       err = registrar(name, atoi(anio), teléfono);
                       res_err = (unsigned char) err;
                       write(s_local, &res_err, 1);
                       break;

          case 1:      read(s_local, teléfono, 4);
                       err_res = buscar(teléfono, name, &anio);
                       write(s_local, name, 4);
                       sprintf(anio_string, "%d", anio);
                       write(s_local, anio_string, 4);

                       res_err = (unsigned char) err;
                       write(s_local, &res_err, 1);
                       break;

          case 2:      read(s_local, name, 1024);
                       read(s_local, teléfono, 12);

                          err = borrar(name, teléfono);
                       res_err = (unsigned char) err;
                       write(s_local, &res_err, 1);
                       break;

          case 3:      read(s_local, name, 1024);
                       read(s_local, teléfono, 12);
                       read(s_local, teléfono_new, 12);

                          err = actualizar(name, teléfono, teléfono_new);
                       res_err = (unsigned char) err;
                       write(s_local, &res_err, 1);
                       break;

          default:
                       break;
      }
      close(s_local);

      pthread_exit(NULL);
}
```

e) Un posible fichero .x sería el siguiente:

```
struct  registrar_args{
    string nombre<1024>;
    int anio;
    string teléfono<12>;
};

struct buscar_res {
   string nombre<1024>;
   int anio;
};

struct borrar_args {
   string nombre<1024>;
   string teléfono<12>;
};

struct actualizar_args{
   string nombre<1024>;
   string old_teléfono<12>;
   string new_telefono<12>;
};

program Telefonos {
    program TelefonosVer {
        int registrar(struct registrar_args) = 1;
        struct buscar_res buscar(string nombre<1024>) = 2;
        int borrar(struct borrar_args) = 3;
        int actualizar(struct actualizar_args) = 4;
   } = 1;
} = 1000;
```

Ejercicio 9. Se desea desarrollar un sistema de monitorización de los nodos de un cluster. En cada nodo del cluster ejecuta un servicio de monitorización, que acepta dos operaciones enviadas desde un nodo central que se utiliza como monitor:

- *start*: que inicia el proceso de monitorización en el nodo.
- *stop*: que para el proceso de monitorización en el nodo.

Una vez arrancado el proceso de monitorización en un nodo, este enviará cada segundo al monitor central la siguiente información: número de procesos arrancados en la máquina, porcentaje de memoria libre, temperatura del procesador, operaciones de E/S realizadas en el último segundo. Cada vez que el monitor central recibe información de un nodo registra dicha información en una base de datos utilizando un servicio denominado registrar, que tiene la siguiente interfaz:

- Registrar(char *IP, int num_procesos, float mem_libre, float temp, int iop)

Donde IP representa la IP del nodo del cluster, num_procesos el número de procesos arrancados en la máquina, mem_libre el porcentaje de memoria libre en el nodo, temp la temperatura del procesador e iop el número de operaciones de E/S realizadas en el último segundo.

Considerando que se quiere desarrollar la aplicación utilizando sockets, se pide:

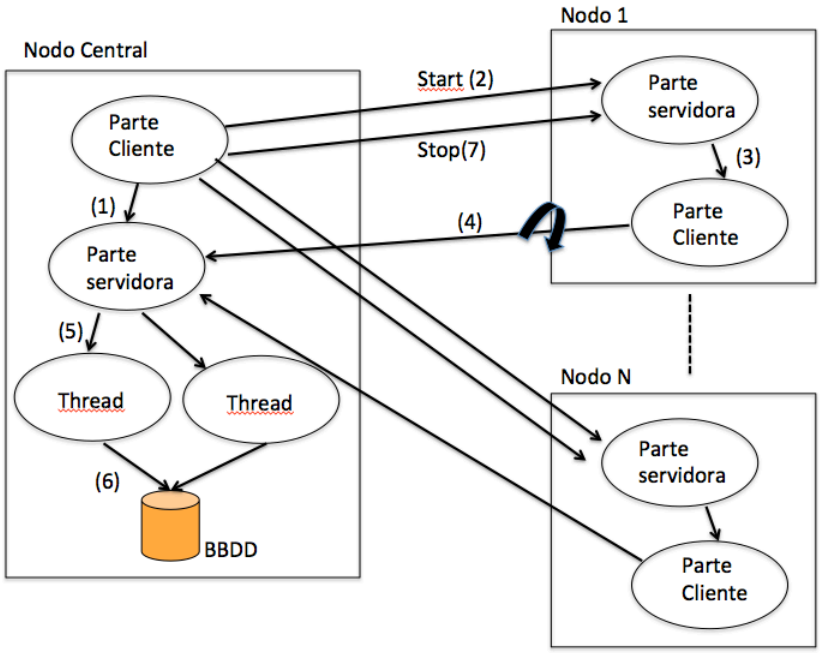
a) Haga un diseño de la aplicación, haciendo todas las consideraciones que crea oportunas.
b) En función del diseño anterior, especifique el pseudocódigo del nodo central que realiza la monitorización de las máquinas y registra la información en la base de datos.

Solución

a)

Como protocolo de transporte se va a utilizar TCP, que ofrece un esquema orientado a conexión que simplifica el desarrollo de la aplicación. Se asume que la dirección IP del nodo central es conocida por todos los nodos del cluster y que el nodo central conoce las direcciones IP de cada nodo del cluster.

Un posible esquema del diseño es el que se muestra en la siguiente figura.



El nodo central que realiza la monitorización creará en primer lugar un proceso ligero (1) que se encargará de recibir los datos de monitorización de los diferentes nodos del cluster. Este proceso ligero hará de servidor. Para cada uno de los nodos que se deseen monitorizar, la parte cliente del nodo central, cada vez que desee monitorizar un nodo, enviará a un proceso servidor que ejecuta en cada nodo un mensaje indicando que desea comenzar la monitorización (2 de la figura). Para ello:

1) Se establecerá una conexión con el servidor que ejecuta en un nodo determinado del cluster, cuya IP y puerto se consideran conocidas (servicio connect).
2) Se enviará por la conexión establecida los siguientes datos:
    a. Código de operación (byte) con valor 0, que indica start.
    b. Puerto en el que el servidor que recibe datos en el nodo central espera las conexiones para recibir los datos de cada nodo del cluster (el nodo central elegirá como puerto el primero libre). El puerto se envía en formato de red.
3) Se cerrará la conexión (servicio close).

Cuando el servidor que ejecuta en cada nodo del cluster recibe el mensaje start, crea un proceso ligero (etapa 3 de la figura) al que se le pasará la IP (que se considera conocida) y el puerto del proceso servidor que recibirá los datos en el nodo central. Cada segundo este proceso enviará los datos de monitorización de la siguiente forma (etapa 4 de la figura):

1) Se establecerá una conexión con el servidor que ejecuta en un nodo central (servicio connect).
2) Se enviarán los datos de monitorización con el siguiente formato:
    a. Dirección IP
    b. Numero de procesos
    c. Memoria libre
    d. Temperatura
    e. IOP

   El formato de cada uno de estos elementos será una cadena de caracteres finalizada con '\0'.
4) Se cerrará la conexión (servicio close).

El proceso servidor del nodo central estará en un bucle infinito esperando conexiones por parte de los diferentes nodos del cluster. Cada vez que acepta una nueva conexión crea un proceso ligero (etapa 5 de la figura) que recibe los datos indicados anteriormente, los convierte al formato adecuado para la llamada Registrar e invoca dicho servicio (etapa 6). A continuación el proceso ligero que atiende la petición finaliza su ejecución.

Cuando se desea finalizar la monitorización de un nodo (etapa 7):

1) Se establecerá una conexión con el servidor que ejecuta en un nodo determinado del cluster, cuya IP y puerto se consideran conocidas (servicio connect).
2) Se enviará por la conexión establecida los siguientes datos:
    a. Código de operación (byte) con valor 1, que indica stop.
3) Se cerrará la conexión (servicio close).

Cuando el nodo servidor del cluster recibe el mensaje stop, finaliza la ejecución el proceso ligero que está enviando datos de monitorización cada segundo.

b)

```
main() {

	short puerto;
	char op;
	int s;
	int sd;

	// parte que realiza el start
	s = socket(…);	// se crea el socket para realizar la conexión

	sd = socket(…)	// crear un socket para el proceso ligero (parte servidora)

	//Buscar un puerto libre para el proceso que recibe los datos de monitorización
	while (bind (sd, IP_nodo_central, puerto) !=0)
		puerto++;

	Crear un proceso ligero (sd)	// pasarle como argumento el descriptor de socket

	Connect()	// conectarse con el nodo del cluster
```

```
	op = 0;			// start

	write(s, &op, 1);		// se envía el código de operación
					// en una implementación real tratar errores
	puerto = htons(puerto);		// pasar a formato de red

	write(s, &puerto, 2);		// se envía el puerto

	close();			// se cierra la conexión


	// parte que realiza el stop
	s = socket(…);	// se crea el socket para realizar la conexión

	Connect()	// conectarse con el nodo del cluster

	op = 1;			// stop

	write(s, &op, 1);		// se envía el código de operación
					// en una implementación real tratar errores

	close();			// se cierra la conexión

}


// pseudocódigo del proceso ligero que recibe la información de cada nodo
Proceso ligero(sd) {

	int sl;
	while(1) {
		sl = accept(sd,…. );	// esperar conexión

		Crear proceso ligero (tratarpeticion, sl);	// pasar como argumento el
								// descriptor de conexión
	}
}


// pseudocódigo del tratarpeticion

void tratarpeticion(int sl){
	char	IP[32]:
	char	mensaje[256];
	int	numProcesos;
	float	memLibre;
	float	temp;
	foat	iop;

	ReadLine(sl, mensaje, 256);		// se recibe la dirección IP
	strcpy(IP,  mensaje);

	ReadLine(sl, mensaje, 256);		// se recibe el número de procesos
	numProcesos = atoi(mensaje);

	ReadLine(sl, mensaje, 256);		// se recibe la memoria libre
```

```
        memLibre = atof(mensaje);

        ReadLine(sl, mensaje, 256);            // se recibe la temperatura
        temp = atof(mensaje);

        ReadLine(sl, mensaje, 256);            // se recibe iop
        iop = atof(mensaje);

        Registrar(IP, numProcesos, memLibre, temp, iop);


        Pthread_exit(0);
}
```

donde `Readline` de un descriptor de socket una cadena de una longitud máxima de n bytes. Si se sobrepasa este número, el resto de los caracteres se descarta. La función devuelve una cadena de caracteres que finaliza con el código ASCII 0. Su código es el visto en el primer laboratorio de la asignatura.

```
ssize_t readLine(int fd, void *buffer, size_t n)
{
     ssize_t numRead;  /* num of bytes fetched by last read() */
     size_t totRead;  /* total bytes read so far */
     char *buf;
     char ch;


     if (n <= 0 || buffer == NULL) {
          errno = EINVAL;
          return -1;
     }

     buf = buffer;
     totRead = 0;

     for (;;) {
          numRead = read(fd, &ch, 1);    /* read a byte */

          if (numRead == -1) {
                    if (errno == EINTR)  /* interrupted -> restart read() */
                         continue;
               else
               return -1;      /* some other error */
          } else if (numRead == 0) {     /* EOF */
                    if (totRead == 0)    /* no byres read; return 0 */
                         return 0;
               else
                         break;
          } else {            /* numRead must be 1 if we get here*/
                    if (ch == '\n')
                         break;
                    if (ch == '\0')
                         break;
                    if (totRead < n - 1) {         /* discard > (n-1) bytes */
                    totRead++;
                    *buf++ = ch;
```

```
                }
            }
        }

        *buf = '\0';
        return totRead;
}
```

Ejercicio 10. Se desea implementar una aplicación de suscripción de video online llamada NetVideo. El sistema está compuesto por dos funcionalidades básicas: suscripción de servicios y visualización de videos. El sistema de suscripción de video permite a los usuarios registrados acceder al sistema de visualización de contenido multimedia. Para ello se disponen de las siguientes llamadas de procedimientos remotos:

- `int login (char *usuario, char *contraseña, int *token)`: Este procedimiento remoto permite gestionar la autenticación de los clientes al sistema. Para ello se pasarán por argumento el identificador de usuario y contraseña. En caso de que el usuario no exista se devuelve 0, en caso contrario se devolverá un token de autenticación. El token es un número entero dentro del rango 10.000 a 60.000.
- `int solicitar_acceso (int token, char* contenido, long *tam, int *puerto)`: Este procedimiento remoto permite averiguar si el contenido solicitado es accesible para dicho usuario. En caso de que sea posible la llamada devuelve el valor 1 y en el parámetro por pasado referencia el puerto del servicio creado bajo demanda para ese usuario. En el argumento `tam`, se devuelve el tamaño que ocupa el video a descargar.

Por otro lado, una vez garantizado el acceso al contenido, los clientes pueden acceder a la visualización del contenido multimedia solicitado. Para ello los clientes podrán acceder directamente al puerto indicado en la llamada `acceso`. Los clientes deben de acceder a la información en función del ancho de banda disponible y la saturación del servidor. Para ello se define el siguiente servicio:

- `int descargaBloqueVideo (int token, int offset, char* buffer, int actBloqueTam, int *sigBloqueTam)`: Esta llamada permite leer `actBloqueTam` bytes del servidor y almacenarlos en el puntero de memoria `buffer`. El parámetro `offset` define la posición de inicio del bloque accedido. Además el servidor informará a los clientes del siguiente tamaño disponible para la siguiente petición mediante el parámetro `sigBloqueTam`. Considere que el tamaño inicial del bloque es de 1 Kbytes.

Los servicios de suscripción y de acceso al contenido correrán dentro del mismo espacio de memoria.

Considerando que se quiere desarrollar la aplicación utilizando sockets en lenguaje C, se pide:

a) Represente los tres servicios presentados anteriormente mediante XDR de SUN.
b) Haga un diseño detallado de la aplicación, haciendo todas las consideraciones que crea oportunas, teniendo en cuentas las características de cada uno de los servicios ofrecidos, con el objetivo de maximizar el rendimiento en la reproducción de video online.

c) Se pide implementar los siguiente servicios mediante el API de socket en lenguaje C:
    a. El servicio `login` en el lado del cliente.
    b. El servicio `descargaBloqueVideo` en la parte cliente. Para ello será necesario acceder al contenido completo.

c. Para mejorar el consumo de la red, se propone hacer una descarga en paralelo del video utilizando el servicio `descargaBloqueVideo` implementado en el apartado b, usando 4 hilos. Cada hilo accederá a una parte del vídeo. Proponga un diseño e impleméntelo.

Solución:

a)

```
struct login_res{
       int código;
       int token
};

struct acceso_res{
       int código;
       int tamaño;
       short puerto;
};

struct bloque_res{
       int código;
       char buffer<>;
       int sigBloqueTam;
};


program NetVideo {
       version NetVideo_V1{
          login_res login (char usuario<>, char contraseña<>) = 1;
          acceso_res solicitar_acceso (int token, char contenido<>) = 2;
           bloque_res descargaBloqueVideo (int token, int offset,
                                           int actBloqueTam)= 3;
       } = 1;
} = 10005;
```

b) El sistema está compuesto por dos tipos de servicios: uno de suscripción y otro de streaming de video. El sistema de suscripción y consulta de videos se tramitará por TCP, ya que se transmitirá información personal y los usuarios pueden estar conectados desde grandes distancias. Por otro lado, el servicio de descarga de bloques ejecutará mediante UDP ya que es fundamental el rendimiento y el orden de los paquetes no es importante.

El sistema estará compuesto por 2 clientes, que invocarán los servicios descritos en el enunciado, y dos servidores, uno para proporcionar el servicio de suscripción y otro destinado a lanzar los servicios de streaming bajo demanda.

Los servidores son basados en sesión ya que hay que almacenar los token dados a cada uno de los usuarios del servicio.

El servidor de suscripción será multithread para poder dar soporte a múltiples clientes concurrentes. El servidor de streaming será inicialmente monothread ya que solo un cliente accederá a ese servicio a través del puerto devuelto en `solicitar_acceso`.

Ambos servicios está localizados en el mismo equipo por lo que comparten IP. El servicio de suscripción usará el puerto 8080.

Los mensajes intercambiados por los servicios serán:

MAX_STRING 50

Mensaje login_pet
Byte código (0)
char usuario[MAX_STRING]
char contraseña[MAX_STRING]

Tamaño = 100 bytes

Mensaje login_res
int resultado
int token

Tamaño = 8 bytes

Mensaje acceso_pet
Byte código (1)
int token
char contenido[MAX_STRING]

Tamaño = 54 bytes

Mensaje acceso_res
int puerto
long tam
int resultado

Tamaño = 12 bytes

Mensaje descarga_pet
int token
int offset
int actBloqueTam

Tamaño = 12 bytes

Mensaje descarga_res
int resultado;
int sigBloque;

Tamaño = 8 bytes + contenido

Se asumirá que los enteros son de 32 bits en formato de red.

c) 1. Login

```
#define MSG_P_TAM 100
#define MSG_R_TAM 8
 int login (char *usuario, char *contraseña, int *token) {

      int sockd;

      struct sockaddr_in serv_name;
      struct hostent *hp;
      int err;
      char  msg_p[MSG_P_TAM];
      char  msg_r[MSG_R_TAM];
```

```
char código = 0;

int srespuesta, int stoken;
/* socket socket creation */

sockd = socket(AF_INET, SOCK_STREAM, 0);
if (sockd == -1){
      perror("Error in socket creation");
      exit(1);
}

/* server address*/
bzero((char *)&serv_name, sizeof(serv_name));
hp = gethostbyname ("netvideo.com");

memcpy (&(serv_name.sin_addr), hp->h_addr, hp->h_length);
serv_name.sin_family = AF_INET;

serv_name.sin_port = htons(8080);
/* connection process */

status = connect(sockd, (struct sockaddr*)&serv_name, sizeof(serv_name));
if (err == -1)
{
      perror("Error al connectar");
      return(-1);
}

bzero (msg_p, MSG_P_TAM);

memcpy(msg,usuario,MAX_STRING);
memcpy(msg + 50,contraseña,MAX_STRING);

código = 0;
if ( write(sockd, &codigo, 1) < 0) {
      perror("Error en enviar peticion");
      close(sockd);
      return (-1);
}


if ( write(sockd, msg, MSG_P_TAM) < 0) {
      perror("Error en enviar peticion");
      close(sockd);
      return (-1);
}

if ( read(sockd, msg_r, MSG_R_TAM) < 0) {
      perror("Error en recibir peticion");
      close(sockd);
      return (-1);
}

close(sockd);

scanf(msg_r,"%d%d",&srespuesta,&stoken);
```

```
        srespuesta = ntohs(respuesta);
         stoken = ntohs(stoken);

        if (srespuesta == 0) {
              return 0;
        } else {
              *token = stoken;
              return srespuesta;
        }
}
```

2. descargarBloqueVideo

Vamos a asumir que el puerto que devuelve la llamada solicitar_acceso se encuentra almacenada en la variable global:

```
short puerto;

#define MSG_P_TAM 12
#define MSG_R_TAM 8

int descargaBloqueVideo (int token, int offset, char* buffer,
                  int actBloqueTam, int *sigBloqueTam) {



        struct sockaddr_in server_addr, client_addr;
        struct hostent *hp;
        ssize_t recv;

        char  msg_p[MSG_P_TAM];
        char  msg_r[MSG_R_TAM];

        int resultado;
        int sigBloque;


        s =  socket(AF_INET, SOCK_DGRAM, 0);
        hp = gethostbyname (“netvideo.com”);
        bzero((char *)&server_addr, sizeof(server_addr));

        memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(puerto);

        bzero((char *)&client_addr, sizeof(client_addr));
        client_addr.sin_family = AF_INET;
        client_addr.sin_addr.s_addr = INADDR_ANY;
        client_addr.sin_port = htons(0);

        bind (s, (struct sockaddr *)&client_addr, sizeof(client_addr));

        bzero (msg_p, MSG_P_TAM);
        sprintf(msg_p,”%d%d%d”, htons(token), htons (offset), htons(actBloqueTam));
```

```
            sendto(s, msg_p, MSG_P_TAM, 0,
            (struct sockaddr *) &server_addr, sizeof(server_addr));


            recvfrom(s, msg_r, sizeof(int), 0, NULL, NULL);

            scanf(msg_r,"%d%d",&resultado,&sigBloque);

            resultado = ntohs(resultado);
            *sigBloqueTam = ntohs(sigBloque);

            trecv = 0;
            while (trecv < actBloqueTam) {
                  recv = recvfrom(s, buffer + trecv, pend, 0, NULL, NULL);
                  trecv += recv;
            }

            close(s);

            return resultado;
      }
```

3. La solución consistirá en dividir el contenido en cuatro bloques contiguos con el objetivo de paralelizar la descarga del contenido. Para ello se puede usar una distribución de este estilo, para que cada hilo descargue el siguiente bloque a reproducir:

Contenido

T0	T1	T2	T3	T0	….

Se considera que el tamaño de bloque no varia durante la retransmisión.

```
pthread_mutex_t mutex_mensaje;
int mensaje_no_copiado = TRUE;   /* TRUE con valor a 1 */
pthread_cond_t cond_mensaje;

#define TAM_BLOQUE 1024
#define NUM_HILOS 4

struct peticion {
      int id;
      char *buffer;
      int tamano;
      int token;
}

void tratar_bloques(struct peticion *p) {
      int j;
```

```
	int sig;
	struct peticion tp;
	/* el thread copia el mensaje a un mensaje local */
	pthread_mutex_lock(&mutex_mensaje);
	memcpy(&tp, p, sizeof(struct peticion));
	/* ya se puede despertar al servidor*/
	mensaje_no_copiado = FALSE;		/* FALSE con valor 0 */
	pthread_cond_signal(&cond_mensaje);
	pthread_mutex_unlock(&mutex_mensaje);

	for (j = 0; j < tp.tamano / (4 * TAM_BLOQUE * NUM_HILOS); j++) {
	    int offset;

	    offset =  (j * NUM_HILOS * TAM_BLOQUE) + (tp.id * TAM_BLOQUE);
	    descargaBloqueVideo (tp.token, offset , tp.buffer + offset, TAM_BLOQUE, &sig);
	}

	pthread_exit(0);
}



int descargaParalela(int token, char * buffer, int tamano)
{
	int i =0;
	pthread thid;
	struct mq_attr q_attr;
	pthread_attr_t t_attr;

	pthread_mutex_init(&mutex_mensaje, NULL);
	pthread_cond_init(&cond_mensaje, NULL);
	pthread_attr_init(&attr);

	for (i = 0; i < NUM_HILOS; i++) {
		struct peticion p;

		p.id = i;
		p.tamano = tamano;
		p.buffer = buffer;
		p.token = token;
		pthread_create(&thid, &attr, tratar_bloques, p);
		while (mensaje_no_copiado)
			pthread_cond_wait(&cond_mensaje, &mutex_mensaje);
		mensaje_no_copiado = TRUE;
		pthread_mutex_unlock(&mutex_mensaje);

	}

}
```

Ejercicio 11 La empresa *Panelillos S.A.* está especializada en el desarrollo e instalación de placas solares para grandes superficies. El CEO de la compañía quiere sacar al mercado una nueva solución para mejorar la eficiencia de absorción de rayos solares. Para ello, se quiere implementar un sistema distribuido que permita la gestión autónoma de las placas solares. El sistema estará compuesto por los siguientes elementos:

- Placas solares: cada placa solar dispone un computador autónomo que contiene un sensor que recoge una serie de datos ambientales: temperatura, humedad, luminosidad y niveles de radiación ultravioleta (estos valores se representarán como números en coma flotante en simple precisión), y un actuador que se encarga de mover la placa. El computador se encarga de leer los datos del sensor, descrito anteriormente, y enviará de forma periódica estos datos a una estación central. Para reducir costes se plantea desplegar una red inalámbrica para poder comunicar las placas y la estación central.

- Estación central: consistirá en un computador que recibirá los datos del computador de cada placa solar y monitorizará el estado de las placas solares con el objetivo de maximizar la energía generada. Por ello, el sistema contará con un sistema inteligente que moverá las placas solares en la orientación más adecuada. La estación central puede realizar dos tipos de llamadas sobre una placa solar dada:
  - Obtener_Energia: mediante esta llamada se obtendrá la energía generada por una determinada placa solar (valor entero).
  - Girar_Placa: mediante esta llamada se podrá mover una determinada placa sobre dos ángulos (coronal y sagital). Los ángulos vendrán dados por números enteros.

  La estación central ajustará la orientación de la placa solar de forma periódica cada 5 minutos. Para obtener la nueva posición la estación hará uso de los datos ambientales recibidos de cada placa. La estancia central dispone de un método local, denominado Predice_Angulo, que recibe los parámetros ambientales y el valor de energía de una placa y devuelve los dos ángulos, que permiten girar la placa.

Considerando que se quiere desarrollar la aplicación utilizando sockets, se pide:

a) Haga un diseño detallado del sistema, haciendo todas las consideraciones que crea oportunas, teniendo en cuenta las características de cada uno de los componentes ofrecidos.
b) De acuerdo al protocolo diseñado, se pide implementar (código C) el siguiente servicio mediante el API de sockets: el código que permite desde el computador central realizar la invocación del método Girar_Placa sobre una determinada placa.

Solución:

a) Para la implementación se ha tendido en cuentas los siguientes aspectos de diseño:

- Paradigma de comunicación

Se aplicará un modelo cliente-servidor, donde tanto los paneles como la estación central serán a la vez clientes y servidores. Por un lado los paneles solares son parte activa ya que envían de forma periódica datos a la estación central, pero a su vez es servidor porque recibe peticiones creadas por la estación central. Del mismo modo, la estación central actúa como un servidor al recibir y almacenar los datos ambientales y además tiene un rol de cliente al interactuar de forma activa con cada una de las placas solares.

- Tipo de conexión

Se ha optado por usar un mecanismo de comunicación basado en UDP, principalmente debido a que los paquetes de datos son pequeños (como se verá al final de este apartado) y se considera que habrá muchos paneles, por lo que se busca reducir en medida de lo posible la cantidad de datos que viaja por la red inalámbrica.

- Concurrencia

Las placas solares están compuestas por un servidor secuencial ya que solo será necesario atender peticiones de un único componente, la estación central. La estación central será un servidor concurrente para que puede recibir datos de forma simultanea por más de una placa solar.

- Estado

La estación central será un servidor con estado ya que es necesario almacenar los datos ambientales de cada placa solar con el objetivo de predecir la orientación de cada una de las placas. En los paneles no será necesario almacenar información entre distintas sesiones.

- Nombrado

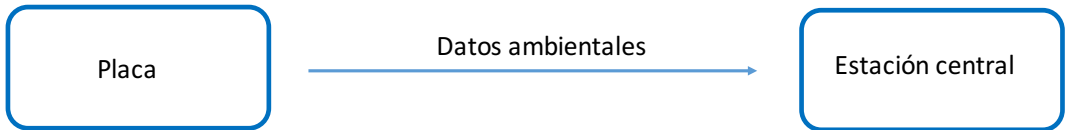
Cada uno de los paneles tendrá asociado una dirección IP dentro del rango 10.0.0.0/24. Además cada panel escuchará peticiones de conexión en el puerto 9090. La estación central tendrá asociada la dirección IP 11.0.0.1 y escuchará peticiones en el puerto 9091.

- Seguridad

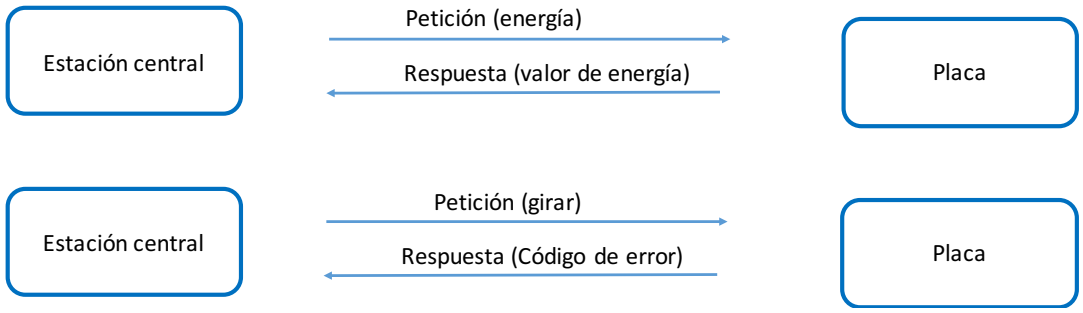
Para evitar sabotajes, se considerará que las conexiones entre los elementos anteriormente descritos serán cifradas.

- Secuencia de intercambio de mensajes

A continuación se describe la secuencia de intercambio de mensajes entre la estación central (servidor) y cada una de las placas (cliente):



La secuencia de intercambio de mensajes entre la estación central (cliente) y las placas (servidor) se muestra a continuación:



- Formato de los mensajes

En el primer intercambio de mensajes (datos ambientales), la placa enviará un datagrama con los siguientes datos:

Temperatura: 4 bytes en formato IEEE 754
Humedad: 4 bytes en formato IEEE 754

Luminosidad: 4 bytes en formato IEEE 754
Radicación: 4 bytes en formato IEEE 754

Tamaño total: 16 bytes por mensaje.

En cuanto al datagrama de petición (energía, girar) estará formado por un datagrama que contendrá los siguientes datos:

Petición: byte (0: obtener energía; 1: girar placa)
Ang1: entero de 32 bits en formato big-endian // se utilizará cuando petición =1
Ang2: entero de 32 bits en formato big-endian // se utilizará cuando petición =1

Tamaño total: 9 bytes por mensaje.

El datagrama de respuesta será:

Valor de energía: entero de 32 bits en formato big-endian
Código de error: byte

Tamaño total: 5 bytes por mensaje.

b) A continuación se mostrará una posible implementación del método distribuido `Girar_Placa`, enmarcado dentro de la estación central. En una implementación real habría que incluir temporizadores para detectar la pérdida de datagramas.

```
#define PUERTO 9090
#define SERLEN 15  // Formato de IP: XXX.XXX.XXX.XXX

int Girar_Placa (  int id  ) { // id corresponde con el identificador de la placa.

   struct sockaddr_in si_panel;
   int                s, i, slen=sizeof(si_panel);
   char               buf[BUFLEN];
   char               servidor[SERLEN]
   int32_t            angulos[2];
   char               mensaje_peticion[9];
   char               mensaje_respuesta[5];
   int32_t            valor_energia;
   char               codigo_error;

   byte               pet_energia = 0;
   byte               pet_girar = 1;

   if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
         perror("socket");

   memset((char *) &si_panel, 0, sizeof(si_panel));
   si_other.sin_family = AF_INET;
   si_other.sin_port = htons(PUERTO);

   sprintf(servidor,”10.0.0.%d”,id);
```

```
    if (inet_aton(servidor, &si_panel.sin_addr)==0) {
           perror("inet_aton() failed\n");
        exit(1);
    }

    memcpy(mensaje_peticion, &pet_energía, 1);
    if (sendto(s, mensaje_peticion, sizeof(mensaje_peticion), 0, &si_panel, slen)==-1)
              perror("sendto()" Comando);
    }

    if   (recvfrom(s,   &mensaje_respuesta,   sizeof(mensaje_respuesta),   0,   &si_panel,
    slen)==-1)
              perror("sendto()" Comando);
    }


   memcpy(&valor_energia, mensaje_respuesta, sizeof(int32_t));
   valor_energia = ntohl(valor_eneria);


    angulos = Predice_Angulo (valor_energia, datos_ambientales[id]);

    agulos[0] = htonl(agulos[0]);
    agulos[1] = htonl(agulos[1]);

    memcpy(mensaje_peticion,   &pet_girar, 1);
    memcpy(mensaje_peticion+1, &agulos[0], sizeof(int32_t));
    memcpy(mensaje_peticion+5, &agulos[1], sizeof(int32_t));


    if (sendto(s, mensaje_peticion, sizeof(mensaje_peticion), 0, &si_panel, slen)==-1)
              perror("sendto()" Comando);
    }

    if   (recvfrom(s,   &mensaje_respuesta,   sizeof(mensaje_respuesta),   0,   &si_panel,
    slen)==-1)
              perror("sendto()" Comando);
    }

    memcpy(&codigo_error, mensaje_respuesta, 1);

    close(s);

    return(codigo_error);

}
```

Ejercicio 12. Se quiere desarrollar un sistema para la notificación de información en estaciones ferroviarias. El sistema consta de un servidor central con información de todas las circulaciones y una serie de estaciones cada una de la cuales recibe información de las circulaciones. La información que incluye una **circulación** es la siguiente:

- Hora de salida de la primera estación.
- Frecuencia de salida de los trenes desde la primera estación (número de minutos entre un tren

y el siguiente).

- Lista de estaciones por las que pasa cada tren. Cada estación viene identificada por un nombre (por ejemplo: “Atocha”).
- Por cada estación se incluye el tiempo de parada del tren en esa estación, en minutos.

El sistema a diseñar sigue un modelo de suscripción. Es decir, cada vez que una estación desea recibir información sobre una determinada circulación, envía un mensaje de suscripción al servidor central. Cada vez que se produce una modificación en una circulación, el servidor central actualiza la información de esa circulación en todas las estaciones que se han suscrito a la misma. Una estación puede estar suscrita a varias circulaciones. Cada **circulación** viene identificada por un número entero.

Cada vez que una estación desea suscribirse a una circulación, lo solicita al servidor central y éste envía como respuesta a este mensaje de suscripción todos los datos asociados a esa circulación. Cada vez que se produce una modificación en una circulación, el servidor central envía los datos de la nueva circulación a todas las estaciones suscritas. El sistema incluye también un servicio para que las estaciones se puedan dar de baja en el sistema de una determinada circulación, de forma que dejará de recibir actualizaciones de dicha circulación. El servidor central también incluye un servicio que permite conocer si existe una determinada circulación y otro que permite conocer el número de estaciones por las que pasa una determinada circulación.

Considerando que se quiere desarrollar la aplicación utilizando sockets, se pide:

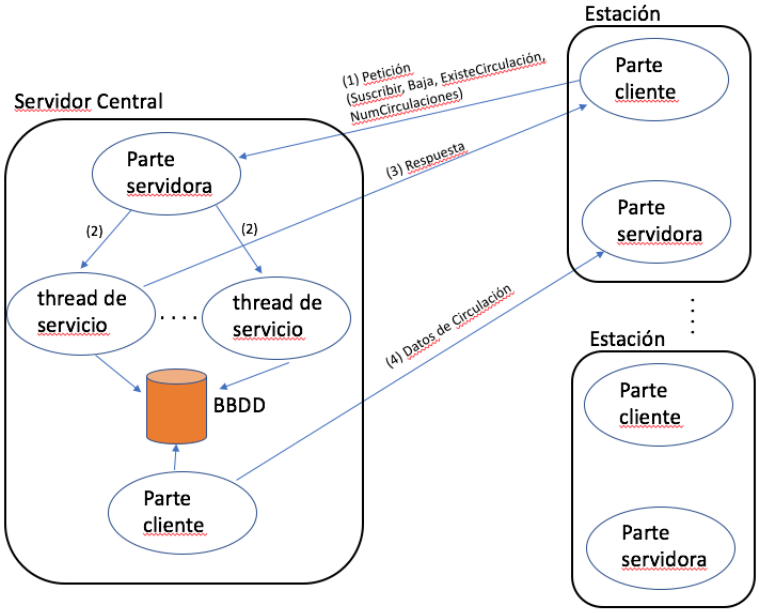
c) Haga un diseño detallado del sistema, haciendo todas las consideraciones que crea oportunas, teniendo en cuenta las características de cada uno de los componentes ofrecidos.
d) Especifique utilizando XDR el procedimiento remoto que permite suscribirse a una determinada circulación.

Solución:

a)

Como protocolo de transporte se va a utilizar TCP, que ofrece un esquema orientado a conexión que simplifica el desarrollo de la aplicación al ofrecer un canal orientado a flujos de bytes y mecanismos que permite conocer la existencia de fallos en la transmisión. Se asume que la dirección IP del servidor central es conocida por todos los equipos instalados en las estaciones. Asimismo, la estación central conoce las direcciones IP de todos los equipos instalados en las estaciones.

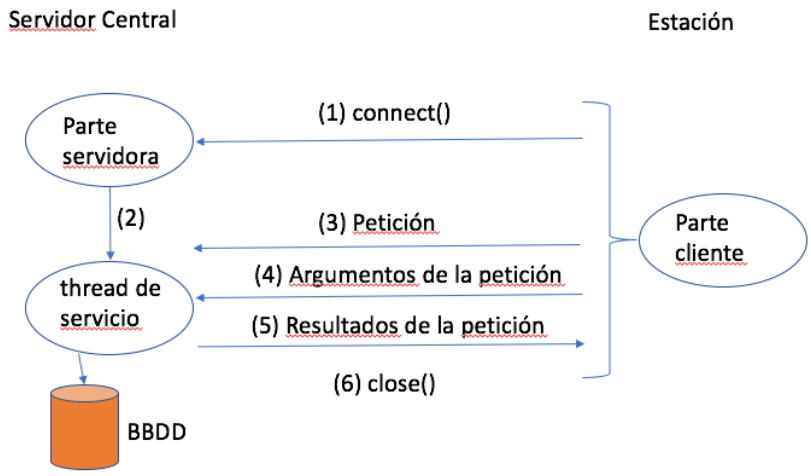
Un posible esquema del diseño es el que se muestra en la siguiente figura.



En cada estación va a haber una aplicación instalada con una parte cliente y otra servidora (dos *threads* de la misma aplicación). Se va a suponer que el puerto utilizado por la parte servidora es el mismo en todas las estaciones, por ejemplo, el puerto 2000. De igual forma, en el servidor central va a haber instalada una aplicación, con una parte servidora (cuyo puerto es conocido, por ejemplo, el 4000) y una parte cliente. La parte servidora de la estación central es la encargada de atender los cuatro servicios que ofrece la aplicación:

- Suscribirse a una nueva circulación
- Dar de baja la suscripción de una circulación
- Conocer si existe una circulación
- Conocer el número de estaciones que hay en una determinada circulación

La parte cliente de la estación es la que se va a encargar de solicitar alguno de estos cuatro servicios. Dado que se utiliza, TCP, el esquema de conexión entre ambas partes va a ser el siguiente, el cual utiliza un modelo de conexión por petición.



Cada vez que la parte cliente de la estación desea solicitar un servicio establece una conexión (1) con la parte servidora que ejecuta en el servidor central (IP del servidor, puerto 4000). A continuación, la parte servidora crea un thread (2) para atender la petición. La parte cliente envía (3) un mensaje identificando la petición (Suscribirse a una nueva circulación, Dar de baja la suscripción de una circulación, Conocer si existe una circulación y Conocer el número de estaciones que hay en una determinada circulación). Una vez enviado este mensaje envía (4) en uno o varios mensajes los argumentos de la petición y obtiene un mensaje con los resultados de la petición (5). Finalmente, ambas partes cierran la conexión (6). Como puede verse la parte servidora de la estación central, se trata de un servidor concurrente que crea un thread para atender cada nueva petición. Este thread tiene acceso a una base de datos donde residirá toda la información que necesita la aplicación.

A continuación, se describe el formato de los mensajes utilizados en esta interacción:

- **Petición** (1). Puede identificarse con un byte:
  - 0 para suscribirse
  - 1 para darse de baja
  - 2 para conocer si existe una circulación
  - 3 para conocer el número de estaciones de una circulación.

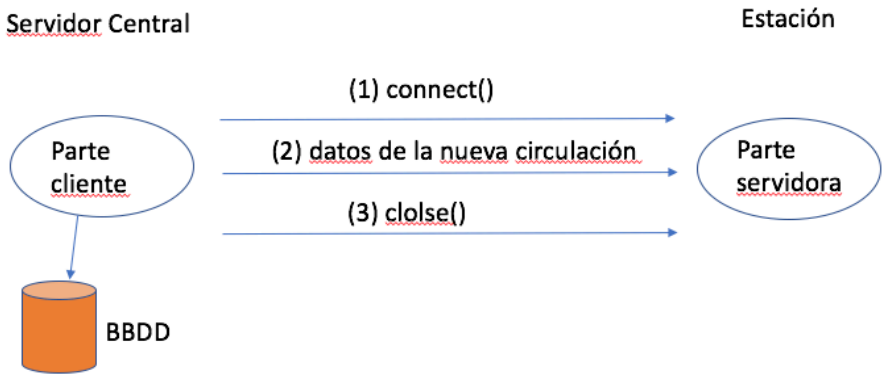
  También podría utilizarse una cadena de caracteres finalizada con el carácter `\0´para identificar cada petición:
  - "ALTA" para suscribirse
  - "BAJA" para darse de baja
  - "EXISTE" para conocer si existe una circulación
  - "NUMERO" para conocer el número de estaciones de una circulación
- **Argumentos para todas las peticiones (2)**. Todas las peticiones envían como argumento el número de circulación. Este puede venir identificado por una cadena de caracteres que identifica la circulación codificada en texto ("234"). La cadena finaliza con el código `\0´para identificar el fin de la cadena.
- **Resultado para el mensaje de suscripción** (3). En este caso en primer lugar se envía al cliente como respuesta, información sobre la existencia o no de la circulación. Esta información se devuelve codificada en un byte (0: existe, 1 no existe). En caso de recibir un 1, se cierra la conexión. En caso de que la circulación exista el cliente recibirá todos los datos asociados a la circulación. Los datos de una circulación incluyen:

  - Hora de salida de la primera estación.
  - Frecuencia de salida de los trenes desde la primera estación (número de minutos entre un tren y el siguiente).
  - Lista de estaciones por las que pasa cada tren. Cada estación viene identificada por un nombre (por ejemplo: "Atocha").
  - Por cada estación se incluye el tiempo de parada del tren en esa estación, en minutos.

  Como formato para todos estos datos se puede enviar una secuencia de cadenas de caracteres, cada una de las cuales finaliza con el código `\0´. Para identificar el fin de la secuencia se pueden utilizar dos códigos `\0´ seguidos. De esta forma, el resultado vendría codificado de la siguiente manera:
  - Una cadena de caracteres con la hora de salida de la primera estación en formato "HH:MM:SS". La cadena finaliza con `\0´.
  - Una cadena de caracteres que codifica la frecuencia en minutos ("240", por ejemplo). La cadena finaliza con `\0´.
  - Por cada estación se envía:
    - Una cadena con el nombre ("ATOCHA"). La cadena finaliza con `\0´.
    - Una cadena que codifica el tiempo de parada en dicha estación ("120", por ejemplo). La cadena finaliza con `\0´.
  - El fin de la secuencia se identifica con dos códigos `\0´seguidos.
- **Resultado para el mensaje de baja** (3). En este caso se puede enviar un simple byte que codifica el resultado de la operación (0: fallo, 1: éxito, etc.).
- **Resultado para la petición que permite conocer si existe una circulación** (3). Se puede devolver un simple byte con el resultado (0: existe, 1: no existe).
- **Resultado para la petición que permite conocer el número de estaciones de una circulación** (3). Se puede enviar una cadena de caracteres ("23"), que codifica este número. La cadena finaliza con el código ´\0´.

Cada vez que una estación se suscribe a una circulación, el servidor guardará el par <número de circulación, dirección IP), para identificar la estación que está suscrita a una determinada circulación. Toda la información que necesita la aplicación residirá en una base de datos accesible por la parte servidora y cliente del servidor central.

De forma desacoplada a estas dos partes, cada vez que hay un cambio en una circulación, la parte cliente del servidor central enviará los datos de la nueva circulación a todas aquellas estaciones suscritas. Para ello, se utilizará una modelo de conexión por cada nuevo envío que seguirá el siguiente esquema:



Cada vez que haya cambios en una circulación que haya que notificar. La parte cliente realizará por cada estación suscrita las siguientes acciones:

- Establecerá una conexión (1) con la parte servidora de la estación (que ejecuta en el puerto 2000 y en una determinada dirección IP).
- Enviará una cadena de caracteres que codifica el número de la circulación (“234”, por ejemplo). La cadena finaliza con `\0´.
- Para el resto de datos de la circulación se seguirá un esquema como el descrito anteriormente. Como formato para todos estos datos enviará una secuencia de cadenas de caracteres, cada una de las cuales finaliza con el código `\0´. Para identificar el fin de la secuencia se pueden utilizar dos códigos `\0´ seguidos. De esta forma, el resultado vendría codificado de la siguiente manera:
  - Una cadena de caracteres con la hora de salida de la primera estación en formato “HH:MM:SS”. La cadena finaliza con `\0´.
  - Una cadena de caracteres que codifica la frecuencia en minutos (“240”, por ejemplo). La cadena finaliza con `\0´.
  - Por cada estación se envía:
    - Una cadena con el nombre (“ATOCHA”). La cadena finaliza con `\0´.
    - Una cadena que codifica el tiempo de parada en dicha estación (“120”, por ejemplo). La cadena finaliza con `\0´.
  - El fin de la secuencia se identifica con dos códigos `\0´seguidos.

e) A continuación, se especifica un posible formato de definición para el servicio de suscripción:

```
struct listaEstaciones {
   string             estación <>;
   int                tiempoParada;
   listaEstaciones    *next;
};

typedef   listaEstaciones *t_lista;

struct restulado {
   string horaSalida<>;
```

```
    int         frecuencia;
    lista       t_lista;
};

program CIRCULACIONES {
     version CIRCULACIONESVER {
          resultado Suscribir(int numCirculacion) = 1;
     } = 1;
 } = 99;
```

Ejercicios de RPC resueltos

Ejercicio 1. Defina las operaciones típicas de un servidor de nombres utilizando como lenguaje de definición de interfaces (IDL), las RPC de ONC.

Solución

```
const MAXPATHLEN = 256;

struct direccion {
      string IP<MAXPATHLEN>;          /* en formato dominio punto */
      int    puerto;
};

program ServidorDeNombres {
     version SDN {
            int          Registrar (string nombre< MAXPATHLEN >,  direccion) = 1;
            direccion    Buscar (string nombre< MAXPATHLEN > ) = 2;
            int          Borrar (string nombre< MAXPATHLEN > ) = 3;
      } = 1;
} = 10000;
```

Ejercicio 2. Se desea implementar un servicio de correo electrónico mediante una arquitectura cliente-servidor. El servicio ofrece la siguiente funcionalidad:

- *Registrar un usuario*: esta función permite a un usuario registrar una cuenta de correo electrónico en un servidor. Un usuario solo puede acceder a las funcionalidades del servidor de correo en caso de estar registrado previamente.
- *Enviar un correo electrónico*: el cliente puede enviar un correo electrónico a otro usuario. En caso de que el destinatario se encuentre registrado en el mismo servidor, almacenará el mensaje en el buzón del destinatario. De lo contrario, el servidor se encargará del reenvío del mensaje al servidor destino correspondiente.
- *Solicitar un correo electrónico*: esta función devuelve el correo electrónico más antiguo almacenado en el servidor, eliminándolo del mismo. En caso de no existir ningún mensaje, el servidor indicará este hecho al cliente.

Los mensajes de correo electrónico contienen los siguientes elementos:

- Nombre del cliente
- Servidor del cliente
- Nombre del destino
- Servidor del destino
- Asunto
- Mensaje
- Adjunto

Todos los campos, excepto el adjunto, son campos de texto. El mensaje puede ser de tamaño variable. Se puede adjuntar el contenido de un único fichero al mensaje, si bien puede aparecer de forma opcional en el mensaje. El tamaño de los datos también es variable.
Se pide:

- Programe utilizando las RPC de SUN, la **interfaz** del servicio entre el cliente y el servidor (*fichero .x*).

- Utilizando el lenguaje C, implemente un programa cliente que permita enviar un correo electrónico (sin adjunto) a un servidor, considerando que se pasan en la línea de mandatos del programa los siguientes campos:
  - Nombre del cliente
  - Servidor del cliente
  - Nombre del destino
  - Servidor del destino
  - Asunto
  - Mensaje

- Utilizando el lenguaje C, implemente un programa cliente que liste los asuntos de todos los mensajes disponibles en el servidor. El programa recibirá en la línea de mandatos los siguientes campos:
  - Nombre del cliente
  - Servidor del cliente

Solución

a)

```
struct mensaje{
   string usuarioOrigen<>;
   string servidorOrigen<>;
   string usuarioDestino<>;
   string servidorDestino<>;
   string asunto<>;
   string mensaje<>
   opaque adjunto<>;
};

struct resultado_msg {
   mensaje msg<>;
};

program MAIL_PRG {
        version MAIL_VER {
                int             registrar    (string usuario<>) = 1;
                int             enviar_mail  (mensaje m) = 2;
                resultado_msg   recibir_mail (string usuario<>) = 3;
        } = 1;
} = 200001;
```

NOTA: se puede utilizar otros tipos de datos para las cadenas de caracteres como: char [*tamaño*], char <>, opaque [*tamaño*] o opaque <>

b)

```
int main(int argc, char *argv[]){
   CLIENT *clnt;
   mensaje mail_1_arg;
   int *result_1;
   char *usuarioOrigen = argv[1];
   char *servidorOrigen = argv[2];
   char *usuarioDestino = argv[3];
   char *servidorDestino = argv[4];
   char *asunto = argv[5];
   char *mensaje = argv[6];

   clnt = clnt_create (servidorOrigen, MAIL_PRG, MAIL_VER, "tcp");
   if (clnt == NULL) {
          clnt_pcreateerror (host);
          exit (1);
   }

   mail_1_arg.usuarioOrigen        = usuarioOrigen;
   mail_1_arg.servidorOrigen             = servidorOrigen;
   mail_1_arg.usuarioDestino             = usuarioDestino;
   mail_1_arg.servidorDestino            = servidorDestino;
   mail_1_arg.asunto               = asunto;
   mail_1_arg.mensaje              = mensaje;
   mail_1_arg.adjunto.adjunto_val  = NULL;
   mail_1_arg.adjunto.adjunto_size = 0;

   result_1 = enviar_mail_1(&mail_1_arg, clnt);
   if (result_1 == (int *) NULL) {
          clnt_perror (clnt, "call failed");
          clnt_destroy (clnt);
          exit (-1);

   }
   clnt_destroy (clnt);
   exit (0);
}
```

c)

```
int main(int argc, char *argv[]){
   CLIENT *clnt;
   mensaje_msg *result_1;
   char *usuarioOrigen = argv[1];
   char *servidorOrigen = argv[2];

   clnt = clnt_create (servidorOrigen, MAIL_PRG, MAIL_VER, "tcp");
   if (clnt == NULL) {
          clnt_pcreateerror (host);
          exit (1);
```

```
}

do{
	result_1 = recibir_mail_1(&usuarioOrigen, clnt);
	if (result_1 == (mensaje *) NULL) {
		clnt_perror (clnt, "call failed");
		clnt_destroy (clnt);
		exit (-1);
	}
	//msg_len indica el número de mensajes que contiene la respuesta
	//msg_len = 0: no hay más mensajes
	//msg_len = 1: hay un mensaje
	//Seria posible enviar todos los mensajes en un único mensaje
} while (result_1->msg. msg_len != 0);

clnt_destroy (clnt);
exit (0);
```

Ejercicio 3. El servidor *Gestor_BD* proporciona dos servicios cuyos prototipos en C son:

int fijar_valor (int item, char valor[]);
int obtener_valor(int item, char valor[]);

En C los vectores se pasan por referencia.
Para poder acceder a estos servicios mediante el mecanismo de RPC, se realiza una especificación del servidor utilizando un hipotético lenguaje de especificación de IDL:

Gestor_BD, version 3
int fijar_valor(in int item, in char valor[20]);
int obtener_valor(in int item, out char valor[20]);

A partir de esta especificación un compilador de IDL genera de forma automática los resguardos del cliente y del servidor. Estos resguardos utilizan las siguientes primitivas de comunicación síncronas para comunicarse entre si:

send(&mensaje, longirud, direccion_destino);
receive(&mensaje, &longitud, &direccion_remitente);

Además se utiliza un esquema de enlace dinámico basado en un binder cuyas operaciones son accesibles a cada máquina mediante una biblioteca que proporciona las siguientes operaciones:

dar_de_alta(nombre_de_servicio, version, direccion_servidor);
dar_de_baja(nombre_servicio,version);
buscar(nombre_de_servicio, version) devueve la direccion del servidor.

Programar en C el código de los resguardos del cliente y del servidor.

Solución:

```
#define FIJAR_VALOR	1
```

```
#define OBTENER_VALOR   2
struct mensaje{
      int funcion;
      int item;
      char valor[20];
      int resultado
};
```

CLIENTE:

```
int fijar_valor(int item, char valor[20])
{
      dir = buscar("Gestor_BD", VERSION);

      m.funcion = FIJAR_VALOR;
      m.item = item;
      memcpy(m.valor, valor, 20);
      send(&m, sizeof(m), dir);
      receive(&m, &long, &dir);
      return(m.resultado);
}

int obtener_valor(int item, char valor[20])
{
      dir = buscar("Gestor_BD", VERSION);

      m.funcion = OBTENER_VALOR;
      m.item = item;
      send(&m, sizeof(m), dir);
      receive(&m, &long, &dir);
      memcpy(valor, m.valor, 20);
      return(m.resultado);
}
```

SERVIDOR:

```
main() {
      dar_de_alta("Gestor_BD", version, dir);

      for(;;) {
            receive(&m, &l, &dir);
            switch (m.funcion) {
                  case FIJAR_VALOR:
                        r = fijar_valor(m.item, m.valor);
                        m2.resultado = r;
                        break;
                  case OBTENER_VALOR:
                        r = obtener_valor(m.item, m.valor);
                        m2.resultado = r;
                        memcpy(m2.valor, valor,  20);
                  break;
            }
            send(&m2. &dor. ...);
}
```

Ejercicio 4. Dada la siguiente definición de interfaz en XDR:

```
typedef int  t_vector_int<>;

program EVALUAR {
        version EVALUARVER {
                int  setVectorInt (t_vector_int v)  = 1;
        } = 1;
} = 99;
```

Indique el prototipo de la función a utilizar en el cliente de las RPC:

Solución:

```
enum clnt_stat setvectorint_1(t_vector_int v, int *r, CLIENT *c);
```

Ejercicio 4. Utilizando el lenguaje de definición de procedimientos remotos XDR, especifique la interfaz asociada a un procedimiento remoto que recibe tres argumentos de entrada (una cadena de caracteres, un número entero y número en coma flotante de simple precisión) y devuelve dos argumentos de salida (una cadena de caracteres y un número entero).

Solución:

```
struct respuesta {
      string      s<>;
      int         a;
};

struct respuesta Procedimiento (string s, int a, float f) = 1;
```

Ejercicio 5. Dada la siguiente definición de interfaz en XDR:

```
typedef int  t_vector_int<>;

program EVALUAR {
      version EVALUARVER {
            int  setVectorInt (t_vector_int v)  = 1;
      } = 1;
} = 99;
```

Indique el prototipo de la función a utilizar en el cliente de las RPC.

Solución:

```
enum clnt_stat setvectorint_1(t_vector_int v, int *r, CLIENT *c);
```

Otros ejercicios resueltos

Ejercicio 1 Responda de forma breve y justificada a las siguientes preguntas:

a) En un servidor NFS, ¿Cuántas RPC son necesarias para realizar la traducción del archivo /usr/users/home/datos.txt Razone su respuesta. Asuma que se dispone del manejador del directorio raíz remoto /.

Hacen falta 4 RPC:
lookup(rootfh, "usr") devuelve (fh0, attr)
lookup(fh0, "users") devuelve (fh1, attr)
lookup(fh1, "home") devuelve (fh2, attr)
lookup(fh2, "datos.txt") devuelve (fh, attr)

b) Considere un esquema de replicación basado en votación (quórum) con 5 servidores (S1,… S5). Si el coste de una operación de escritura supone 3 veces mas que una petición de lectura y el 40 % operaciones son de lectura ¿cuál debería ser el quórum de lectura y escritura para que el sistema sea óptimo? Razone su respuesta.

Los valores de R y W deben cumplir:

$R + W > N$
$W + W > N$
$R, W < N$

El coste asociado a los posibles valores es el siguiente:

R	W	Coste
1	5	1*0,4*1 + 5 * 0.6 * 3 = 9.4
2	4	2*0.4*1 + 4*0.6*3 = 8
3	3	3*0.4*1 + 3*0.6*3 = 6.6

Los valores que minimizan, por tanto, el sistema son R=3 y W=3.

c) Describir brevemente qué es un servicio web.

Un **servicio web** es una colección de protocolos y estándares abiertos que sirven para intercambiar datos entre aplicaciones:

a. Escritas en distintos lenguajes de programación
b. Ejecutan en distintos sistemas operativos y arquitecturas
c. Desarrolladas de manera independiente

Los servicios web son independientes de la aplicación que los usa. Se basa en los siguientes protocolos y estándares: http, SOAP, XML, UDDI y WSDL.

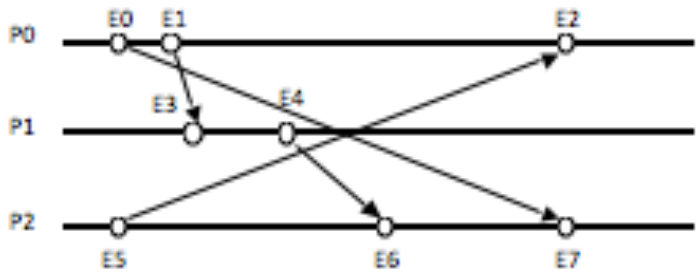
d) Definir brevemente en qué consiste el multicast atómico.

Un multicast es una operación en la que un emisor envía un mensaje a varios procesos receptores. Un multicast atómico es un multicast fiable (el mensaje es recibido por todos los nodos en funcionamiento) que asegura que todos los miembros del grupo recibirán los mensajes de diferentes nodos en el mismo orden.

e) ¿Qué ventajas tiene el uso de una memoria caché en el cliente de un sistema de ficheros distribuido?

- Mejora el rendimiento de las operaciones de lectura y escritura.
- Permite utilizar optimizaciones como lectura adelanta y escritura difereida
- Reduce el tráfico por la red
- Reduce la carga en los servidores
- Mejora la capacidad de crecimiento del sistema.

f) Usando relojes vectoriales especificar las marcas de tiempo de cada uno de los eventos de la figura



E0 = (1,0,0) E1=(2,0,1) E3=(2,1,0) E4=(2,2,0) E5=(0,0,1) E2=(3,0,1)
E6=(2,2,2) E7=(1,0,3)

g) Para qué sirve el s*ervici*o listen()?

Habilita un socket para recibir conexiones en un servidor TCP. El parámetro del servicio especifica el número máximo de peticiones que se pueden encolar antes de ser aceptadas en la operación accept().

Ejercicio 2. 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:

	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.

Solución:

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

No se puede actualizar la replica de la partición {4,5,6} porque no pertenece a una partición mayoritaria.
M = max{NV} = 6
I = {4,5,6}
N = max{SCi} = 3
Si |I| > N/2 -> Si 3 > 3/2 se puede actualizar la partición. Por tanto sí se puede actualizar.

Ejercicio 3. ¿Qué es un multicast atómico? Ponga un ejemplo. Describa un método que permita implementar la ordenación total en un multicast atómico.

Es un tipo de multicast fiable que asegura que todos los miembros del grupo recibirán los mensajes de los diferentes nodos en el mismo orden. Por ejemplo, si en un grupo de 4 procesos dos ellos realizan un multicast al resto (m1, m2), cada miembro recibirá los mensajes en el mismo orden. O bien se recibe m1 y luego m2 en todos, o bien se recibe m2 y luego m1 en todos.

Una forma de implementar la ordenación total en un multicast atómico es mediante un método centralizado, que utiliza un proceso secuenciador para asignar identificadores de orden total (IOT) a los mensajes. Cada mensaje se envía al secuenciador, el secuenciador incrementa el IOT, le asigna un IOT al mensaje y lo envía a todos los procesos. En cada nodo los mensajes se deben ir entregando en orden de IOT, no en orden de recepción.

Ejercicio 4. Dado el siguiente fragmento de código, indique qué problema(s) plantea. Considere que s representa un descriptor de socket correctamente definido.

```
struct mensaje {
      int a;
      float b;
      char c;
};

int n;
struct mensaje m;

m.a = 4;
m.b = 2.3;
m.c = ‘1’;

n = write(s, (const void *) &m, sizeof(struct mensaje));
if (n != sizeof(struct mensaje)) {
      printf(“Error en el envío del mensaje\n”);
```

```
            close(s);
    }
```

El código anterior plantea dos problemas. En primer lugar, no se pueden enviar estructuras de datos de un determinado lenguaje de programación, en este caso C, a través de un socket. Una estructura de datos es dependiente de un lenguaje de programación. Si en el otro extremo de la comunicación, el programa está escrito, por ejemplo, en Java, no sabrá cómo tratar la estructura.
Por otra parte, la llamada write devuelve el número de bytes realmente enviados y no se puede considerar un error, que la llamada devuelva un valor menor del que se desea en principio enviar.

Ejercicio 5. ¿Qué ventajas ofrecen los relojes lógicos vectoriales respecto a los relojes lógicos de Lamport?

Cuando se utilizan relojes lógicos de Lamport, dados dos eventos a y b se cumple:

Si a □ b entonces RL(a) < RL(b)

Donde RL(a) es el valor del reloj lógico asociado al evento a y RL(b) el valor del reloj lógico asociado al evento b.

Con los relojes vectoriales se cumple:

a □ b si y solo si RV(a) < RV(b)

donde RV(a) es el valor del reloj vectorial asociado al evento a y RV(b) el valor del reloj vectorial asociado al evento b.

Ejercicio 6. Se desea diseñar e implementar un servidor de ficheros sin estado, que de soporte a clientes heterogéneos en una red de área local. Este servidor sin estado se diseñará e implementará para ejecutar en el espacio de usuario. La implementación ser va a realizar utilizando sockets. Responda a las siguientes preguntas:

a) Especifique una posible interfaz para una biblioteca, que proporcione acceso al servidor de ficheros, utilizando el lenguaje de programación C, para las siguientes operaciones sobre un fichero: open, close, read, write y delete, que permite borrar un fichero. Considere que las operaciones de lectura y escritura permiten acceder a un bloque de cualquier tamaño.

b) De acuerdo a la interfaz anterior, diseñe el protocolo del servidor de ficheros, de forma que cualquier cliente de la red pueda realizar una implementación de la parte cliente.

c) De acuerdo a los dos apartados anteriores, implemente, utilizando el lenguaje de programación C, la operación read, en el lado del cliente del sistema de ficheros.

d) Especifique la interfaz definida en el apartado a) en XDR.

e) Se quiere completar este servicio de ficheros con un servicio que ofrezca exclusión mutua distribuida. Describa una solución que asegure exclusión mutua distribuida a los clientes que utilizan el servicio de ficheros.

f) Con el objetivo de mejorar la fiabilidad del servicio, se quiere diseñar un servicios de ficheros replicado. Proponga una solución teniendo en cuenta que se quiere asegurar en todo momento la consistencia de los datos.

g) Se desea completar el servicio con el empleo de cachés en los clientes del sistema de ficheros. Para intentar asegurar la coherencia en el acceso a los datos se propone utilizar el modelo de coherencia que emplea NFS. ¿Cómo intenta resolver NFS el problema de coherencia de caché?

h) Dado el siguiente diagrama de eventos que ocurren en el sistema descrito anteriormente, anote mediante relojes lógicos y vectoriales las marcas de tiempo de los siguientes eventos. Además, indique qué eventos son concurrentes entre sí.

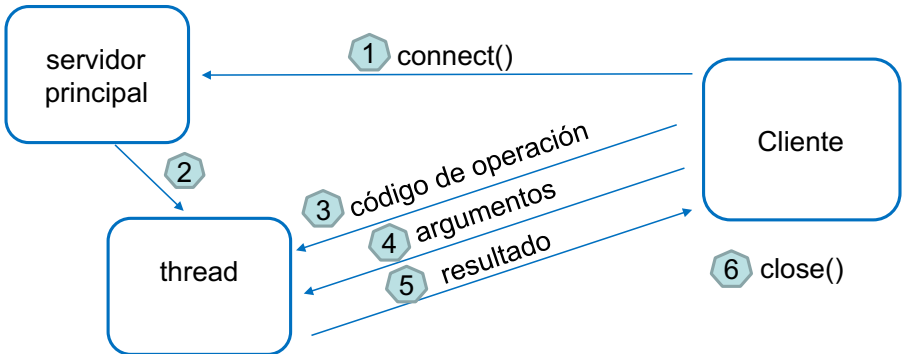
Solución:

a) Puesto que se pretende diseñar un servidor de ficheros sin estado, que ejecute en el espacio de usuario, para clientes heterogéneos, la interfaz a diseñar debe contener operaciones autocontenidas. Una posibilidad es la siguiente. En todas las operaciones se ha incluido como argumento el nombre o dirección IP de servidor y el puerto. De esta forma, la interfaz puede servir para que un cliente pueda interactuar con varios servidores a la vez. Como el servidor de ficheros es sin estado, éste no puede guardar ningún tipo de descriptor asociado a un fichero, por ello todas las operaciones deben incluir el nombre del fichero con el que desean operar.

- `int open(char *name, mode_t mode, char *server, short port)`. La operación devuelve 0 en caso de éxito o -1 en caso de error.
- `int close(char *name, char *server, short port)`. La operación devuelve 0 en caso de éxito o -1 en caso de error.
- `int read(char * name, int offset, int length, char *data, char *server, short port)`. La operación lee del fichero name a partir de una determinada posición (offset). El número de bytes a leer se especifica en length y los datos se almacenan en data. La operación devuelve el número de bytes realmente leídos o -1 en caso de error.
- `int write(char * name, int offset, int length, char *data, char *server, short port)`. La operación escribe en el fichero name a partir de una determinada posición (offset). El número de bytes a escribir se especifica en length y los datos a escribir se almacenan en data. La operación devuelve el número de bytes realmente escritos o -1 en caso de error.
- `int delete(char *name, char *server, short port)`. La operación devuelve 0 en caso de éxito o -1 en caso de error.

b) Se trata de un típico protocolo cliente servidor. Se va a utilizar como protocolo de transporte TCP para asegurar que los datos que se leen o escriben se reciben en orden y poder conocer que los mensajes se han transmitido con éxito o no. Para conseguir tolerancia a fallos se va a utilizar un modo

de conexión por petición. En cada operación se realizará una conexión al servidor, se enviará el código de operación (open, close, read, write y delete), los argumentos, se recibirá el resultado y se cerrará la conexión. El servidor será concurrente y lanzará un thread para atender cada petición.



A continuación se describe el formato asociado a los mensajes 3, 4 y 5.

- Código de operación: un byte:
  - 0: `open`
  - 1: `close`
  - 2: `read`
  - 3: `write`
  - 4: `delete`
- Operación `open`:
  - Argumentos:
    - `name`: cadena de caracteres acabada con el código '\0'.
    - `mode`: entero de 32 bits en formato big-endian
  - Resultado:
    - Código de error: entero de 32 bits en formato big-endian
- Operación `close`:
  - Argumentos:
    - `name`: cadena de caracteres acabada con el código '\0'.
  - Resultado:
    - Código de error: entero de 32 bits en formato big-endian
- Operación `read`:
  - Argumentos:
    - `name`: cadena de caracteres acabada con el código '\0'.
    - `offset`: entero de 32 bits en formato big-endian
    - `length`: entero de 32 bits en formato big-endian
  - Resultado:
    - Código de error: entero de 32 bits en formato big-endian
    - `Datos`: bloque de datos (bytes), dependiendo el tamaño indicado en el campo anterior.
- Operación `write`:
  - Argumentos:
    - `name`: cadena de caracteres acabada con el código '\0'.
    - `offset`: entero de 32 bits en formato big-endian
    - `length`: entero de 32 bits en formato big-endian
    - `Datos`: bloque de datos (bytes), de tamaño length

- Resultado:
  - Código de error: entero de 32 bits en formato big-endian

- Operación `delete`:
  - Argumentos:
    - `name`: cadena de caracteres acabada con el código '\0'.
  - Resultado:
    - Código de error: entero de 32 bits en formato big-endian

c)

```
int read(char * name, int offset, int length, char *data,
                char *server, short port)
{
      int sd;
      struct sockaddr_in server_addr;
      char op;
      int err;
      int32_t n;

      sd = socket(AF_INET, SOCK_STREAM, 0);
      if (sd < 0)
            return (-1);

      bzero((char *)&server_addr, sizeof(server_addr));
      hp = gethostbyname (server);
      memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
      server_addr.sin_family  = AF_INET;
      server_addr.sin_port    = htons(port);

      // se establece la conexión
      if (connect(sd, (struct sockaddr *) &server_addr,  sizeof(server_addr)) <0) {
            close(sd);
            return(-1);
      }

      op = 2; // operación read
      err = write(sd, &op, 1);
      if (err < 0) {
            close(sd);
            return(-1);
      }
      err = write(sd, name, srlen(name)+1);  // nombre de fichero
      if (err < 0) {
            close(sd);
            return(-1);
      }

      n = htonl(offset);
      err = write(sd, &n, sizeof(int32_t));  // offset
      if (err < 0) {
            close(sd);
            return(-1);
```

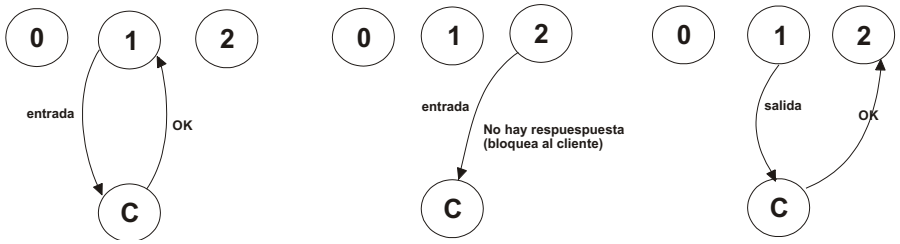
```
        }

        n = htonl(length);
        err = write(sd, &n, sizeof(int32_t));  // length
        if (err < 0) {
                close(sd);
                return(-1);
        }

        err = read(sd, &n, sizeof(int32_t));  // longitud del bloque de datos
        if (err < 0) {
                close(sd);
                return(-1);
        }
        n = htonl(n);
        if (n <= 0){
                close(sd);
                return(n);
        }

        while (n >0) {
                err = read (sd, data, n);
                if (err < 0) {
                        close(sd);
                        return(-1);
        }
        n = n - err;
        data = data + err;
                        }

                        close (sd);

                        return(err);
  }
```

d) Una posible interfaz es:

```
        struct arg_Name {
                string name;
                int    mode;
        };

        struct arg_Read {
                string       name;
                int    offset;
                int    length;
        };

        struct res_Read {
                int    n;
                opaque data<>;
        };

        struct arg_Write {
                string name;
```

```
        int    offset;
        int    length;
        opaque data<>;
};

program FS_SERVER {
        version FS_SERVER_VER {
int open(arg_Name)        = 1;
             int close(arg_Name)              = 2;
             res_Read read(arg_Read)   = 3;
             int write(arg_Write)      = 4;
             int delete(arg_Name)      = 5;
        } = 1;
} = 99;
```

e) Una posible solución es utilizar un servidor que utilice el algoritmo centralizado para asegurar exclusión mutua. Este servidor o coordinador funciona del siguiente modo:



f) Una forma es utiliza el modelo de replicación basado en copia primaria con funcionamiento síncrono. Para hacer frente a K fallos se utilizan K+1 servidores. Uno de ellos actúa de servidor principal. Cuando recibe una petición la propaga el resto de servidores y una vez propagada la operación responde al cliente. Si el servidor principal falla, uno nodo de respaldo toma el papel de servidor principal.

g) Los clientes deben:

- Enviar periódicamente los bloques modificados al servidor
- Cuando un fichero se abre en un cliente se comprueba en el servidor si la información se ha modificado
- Si se ha modificado se descartan los bloques de la caché
- Piden al servidor que revalide la información que tiene en su caché
  - 3-30 segundos para ficheros
  - 30-60 segundos para directorios
  - El bloque se descarta si ha sido modificado en el servidor
- Los atributos en caché se descartan después de
  - 3 segundos para atributos de ficheros
  - 30 segundos para atributos de directorios

h) Utilizando relojes lógicos:

- E1: 1
- E2: 2
- E3: 3
- E4: 1
- E5: 2
- E6: 2
- E7: 3
- E8: 4
- E9: 1
- E10: 5

Mediante relojes vectoriales:

- E1: (1000)
- E2: (2100)
- E3: (3200)
- E4: (0100)
- E5: (0200)
- E6: (0010)
- E7: (0021)
- E8: (0031)
- E9: (0001)
- E10: (0032)

Son concurrentes entre otros:

E1 || E4, E2 || E5, E4 || E6, E4 || E7, E4 ||
E8, E5 || E7, E5 || E8,