Filesystems, IPCs y Servidores Concurrentes

Informe

Arolfo Franco (51408)
Mozzino Jorge (51628)

Introducción

Se nos presento el desafío de implementar una aplicación del estilo “cliente-servidor”, utilizando distintos tipos de IPCs para que ambos procesos se comuniquen. Se trata de un juego donde se cuenta con una aplicación servidor y varios clientes, los cuales generan ligas, equipos y acciones tales como intercambiar jugadores o seleccionar jugadores.

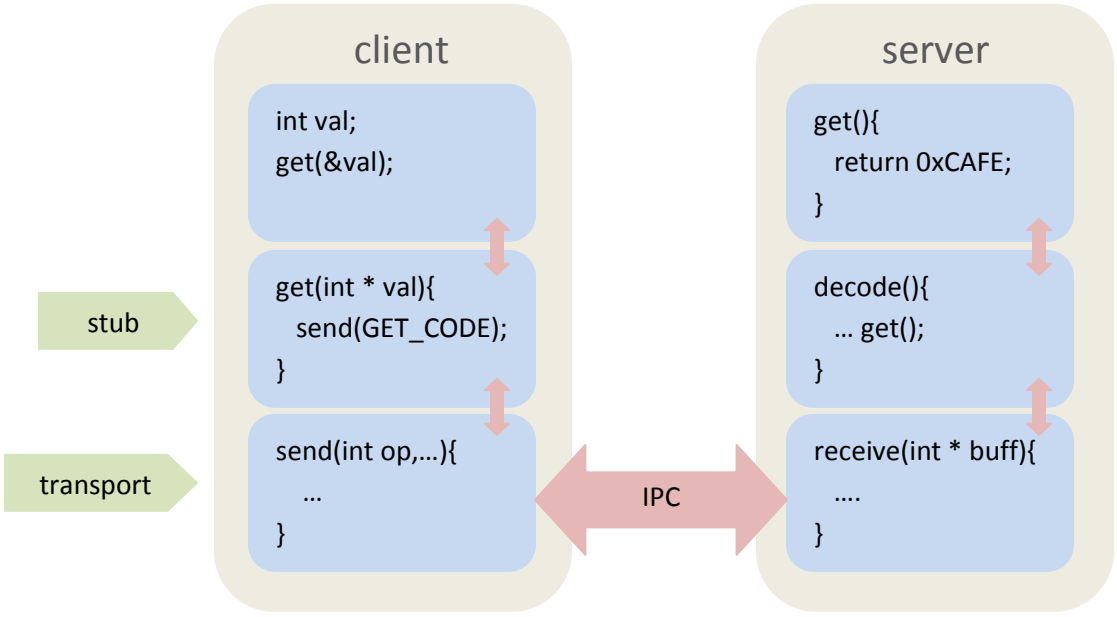
Se presenta con este trabajo 8 aplicaciones, un “server” y un “client” por cada una de las siguientes IPCs:

- FIFOs.
- Message queue.
- Shared Memory.
- Sockets.

Desarrollo

Arquitectura de “cliente-servidor”

Se decidió resolver este trabajo práctico modularizando tanto al “client” como al “server” en 3 partes: programa en sí, el cual es el encargado de la lógica de la aplicación y ejecución de actividades; sector de stub o marshalling, encargado de codificar paquetes de datos que se comunicaran entre ambas aplicaciones; y sector de “transport”, encargado de la comunicación entre ambos procesos.



Así, por ejemplo, supongamos que el cliente quiere un valor que posee el server, así que este llama a la función get(), encargada de obtener dicho valor. Dicha función estará implementada en la sección stub de client, que codificará ese pedido en el envío de un código de operación (u opcode) que le dirá al servidor "quiero el valor de get()". Luego el sector de "transport" es el encargado de enviar el paquete que se codifico con el opcode al programa del servidor. Así mismo, en el "transport" del servidor se recibe el paquete y se deriva al stub del mismo. Este lo decodifica y ve que el programa que envió el paquete requiere la función "get()", así que la ejecuta.

Luego, el servidor enviara un paquete con el resultado de manera análoga a como lo hizo el cliente.

API de IPCs

Para el desarrollo de esta aplicación, se diseño una API de IPCs con funciones que se implementaron en cada caso en particular. Tanto servidor como cliente fueron desarrollados con esta API independientemente de cómo fuera implementada, ya sea socket, FIFO, shared memory, etc.

Las funciones involucradas son las siguientes (esto se puede encontrar en physic.h):

```
   /*API IPCs
      Esta librería maneja la comunicación entre dos programas del estilo
cliente-servidor mediante la inicialización e uso de una variable que hace
referencia al canal. Esta será una variable void * que se inicializa con las
siguientes funciones.
   Es responsabilidad del usuario que haya espacio suficiente para esta
variable.*/

   #define SERVER 0
   #define CLIENT 1

   #define MAX_SIZE 1024 //Máxima dimensión del canal
   #define MAX_DATA 1024 //Máximo de datos a transmitir

   /**
     Inicializa el canal por default.
    */
   int getDefaultChannel(void * channel);

   /**
     Inicializa el canal con un canal nuevo.
    */
   int getNextChannel(void * channel);

   /**
     Crea el canal de transmisión.
     Antes de llamarla se debió obtener el canal, ya sea con
getDefaultChannel, getNextChannel o stringToChannel.
     Se recomienda que sea el servidor quien la llame.
    */
   int createChannel(void * channel);

   /**
     Conecta el canal.
     El parámetro who indica de que programa se trata: CLIENT o SERVER.
```

```
     Deben ser un canal inicializado previamente con createChannel().
    */
   int connectToChannel(void * channel, int who);

   /**
     Si hay algún paquete para leer del canal channel, lo coloca en el
buffer. Si no, espera a que haya algún mensaje en ese canal.
     El parámetro size es el máximo que se puede leer.
     El parámetro who indica quien llama a función, toma el valor SERVER o
CLIENT.
     Retorna el tamaño del mensaje leído.
    */
   int receivePacket(char * buffer, int size, void * channel, int who);

   /**
     Envía una cantidad size de bytes del buffer en el canal channel.
     El parámetro who indica quien llama a la función, toma el valor SERVER o
CLIENT.
    */
   int sendPacket(char * buffer, int size, void * channel, int who);

   /**
     Codifica el canal channel en el buffer.
     Retorna el tamaño del paquete de datos introducido en buffer.
    */
   int channelToString(char * buffer, void * channel);

   /**
     Inicializa el canal channel con los datos del buffer.
     Buffer debe haber sido inicializada por channelToString().
    */
   int stringToChannel(char * buffer, void * channel);

   /**
     Inverso a connectToChannel(), desconecta el canal channel.
    */
   int disconnectFromChannel(void * channel);

   /**
      Inverso a createChannel(), destruye el canal channel.
    */
   int destroyChannel(void * channel);
```

En cada implementación de IPC, se diseño una estructura del tipo Channel, la cual tenía datos que identificaban a cada canal. Así, cada void * no es más que una referencia a una estructura del tipo Channel.

Ahondaremos en sus implementaciones en la sección dedicada a la implementación de IPCs.

Nota: Es responsabilidad del programador que utilice esta API el reservar memoria para cada void * que referencie a un canal.

Implementación "Cliente-Servidor"

Desde el punto de vista del cliente, se decidió dividir al programa en dos procesos mediante la llamada al sistema `fork()`. Antes de que ella ocurra, se inicializan dos pipes de comunicación entre ambos procesos: syncPipe, utilizado para sincronizar ambos programas del cliente, y signalPipe, utilizado para enviar códigos de operación en el manejo de señales.

Se decidió el uso de señales para dos ocasiones: iniciar el shell y durante el draft. Luego se explicara por qué se utilizan señales en esos momentos.

Ahora, se decidió el fork() antes mencionado para que el cliente tenga dos procesos en paralelo, los cuales uno se ocupa de el shell y el envío de datos al servidor, mientras que el otro está en una constante recepción de paquetes del servidor e imprime el mensaje que el servidor ha enviado. Así, tendremos:

```
Pseudocódigo del cliente

Inicialización()
If( fork() ){
      En el caso del hijo
      while(1){
            Recibe un paquete (operación bloqueante)
            Lo muestra
      }
}else{
      En el caso del padre
      while(1){
            shell();
            envió del código de operación al server
      }
}
```

Hemos visto que podía ocurrir el siguiente problema: el cliente se encuentra en el Shell, ejecuta un comando, y este se envía al servidor, pero como el servidor tarda en contestar, el cliente ejecuta shell antes de que el otro proceso pueda imprimir la respuesta del servidor. Así, tendríamos en la consola:

```
:> command
:>
Respuesta de command
:>_
```

Para evitar este problema de sincronización, se inicializó el pipe "syncPipe", así el programa que pide un comando se bloquea antes de pedir otro (read) y el otro escribe en el pipe luego de contestar el mensaje del server.

Ahora tendremos:

```
Pseudocódigo del cliente

Inicialización()
If( fork() ){
```

```
            En el caso del hijo
            while(1){
                  Recibe un paquete (operación bloqueante)
                  Lo muestra
                  Escribe en syncPipe
            }
      }else{
            En el caso del padre
            while(1){
                 shell();
                 envío del código de operación al server
                 Lee de syncPipe
            }
      }
```

Ahora procederemos a explicar algunas implementaciones, en particular el caso del draft.

Se cuenta con el comando joindraft, el cual envía el código de operación de joindraft al server y se cuelga en el read() del pipe. A la par, el otro proceso recibe un paquete del server con el opcode de joindraft exitoso y escribe en el pipe un mensaje particular de draft. Luego vuelve a la recepción de paquetes.

Una vez leído por el proceso que lee del pipe, este va a un shell particular de draft, que solo tiene la opción de salir mediante un quit.

Ahora, el server enviara el opcode IS_ALIVE, el cual me dice que el draft está listo y se debe responder para verificar que todos los usuarios están listos.

Así que el client que está recibiendo datos lee el IS_ALIVE y debe avisarle al otro proceso que llego dicho paquete, pero ese proceso está detenido en un scanf() del shell de draft, así que debemos interrumpirlo de alguna manera. Por este motivo utilizamos señales en el draft.

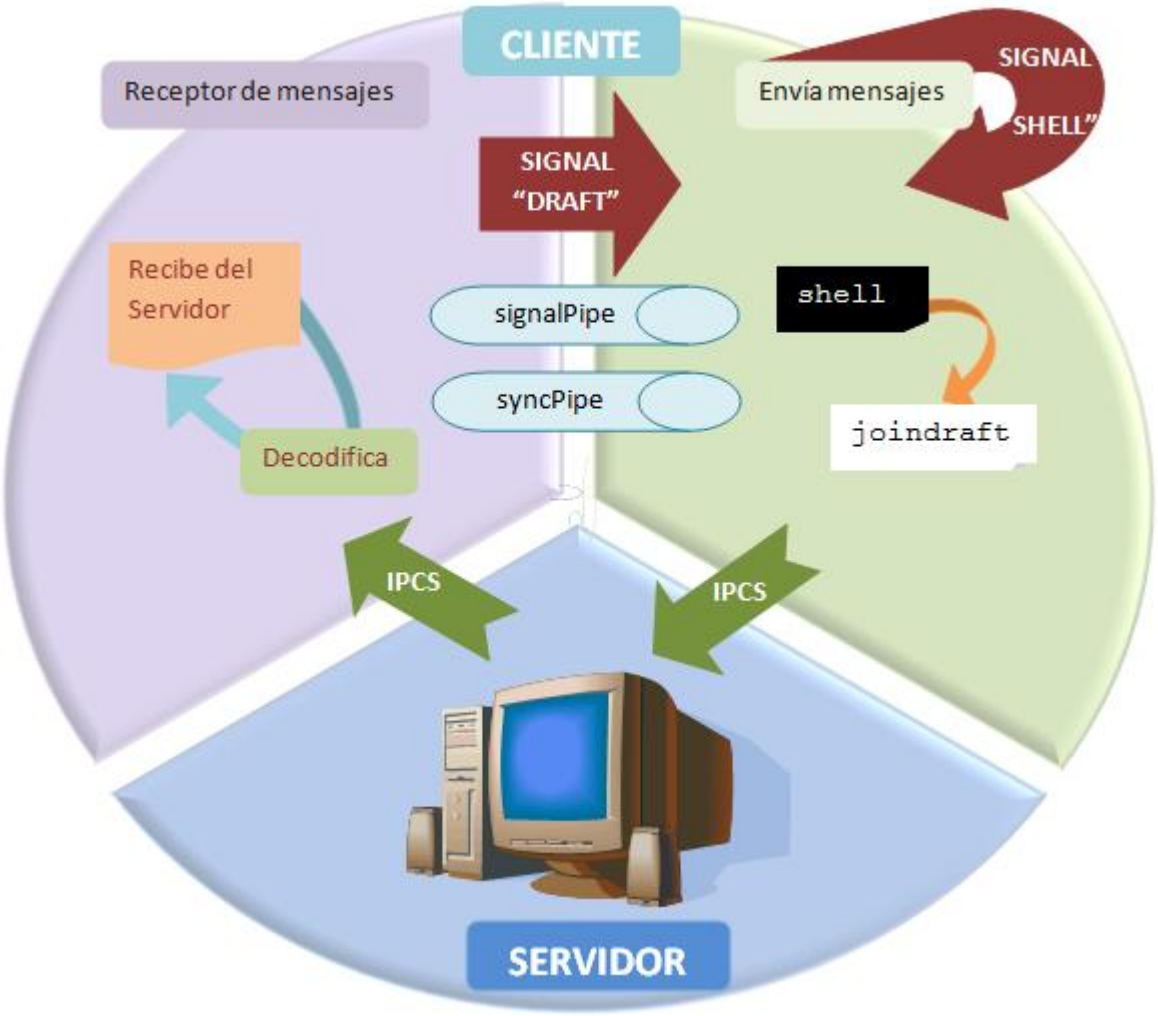
El proceso de receive() envía la señal al otro proceso y escribe en el signalPipe un opcode diciendo que se trata de el IS_ALIVE, pues tenemos otros opcodes durante el draft que requieren el envío de la misma señal.

Por otra parte, el proceso de envío de datos recibe la señal y la atiende en una rutina que lee del signalPipe a ver por qué se envió la señal. Este, en el caso de IS_ALIVE, muestra un mensaje de "estas ahí?" y envía al server un paquete de datos con IS_ALIVE y un número distinto de -1 para indicar que todo está bien.

Luego, el server enviara mensajes de CHOOSE_PLAYER o END_DRAFT, los cuales indican que se debe elegir un jugador o que se termino el draft, respectivamente. Estos mensajes también enviaran la señal de draft.

Finalmente, con END_DRAFT o QUIT_DRAFT se volverá al shell por medio de la señal específica que ejecuta el shell.

Podemos resumir el draft en el siguiente diagrama:



Respecto del servidor, se decidió implementarlo utilizando diversos threads. Al momento de diseñarlo, se presentaron dos opciones: hacer un servidor multi-thread o un servidor multi-proceso (Nótese que era necesario una de estas dos opciones ya que el servidor debía poder atender a varios clientes al mismo tiempo). La razón por la cual se decidió por el servidor multi-thread fue que era más fácil el manejo de los datos. Al compartir los threads la zona de memoria global, se podría trabajar manejando los datos en memoria. En cambio, con un servidor multi-proceso era necesario estar constantemente levantando la información de los archivos y bajándola luego de procesarla. Para evitar cualquier problema de pérdida de información en caso de que el servidor se cayera, se decidió guardar la información cada un segundo.

Además, como ya se ha introducido en el párrafo anterior, el servidor debía realizar otras tareas aparte de atender a los clientes. Entre éstas se encuentran: guardar la información en archivos, verificar que los clientes no hayan perdido la conexión, verificar que no haya expirado el plazo de los *trades *y jugar los partidos al momento en el que se agregan al directorio de partidos.

Para que el servidor pudiera realizar todas estas tareas, fue necesario crear más threads: cada uno especializado en una tarea.

Para atender a los clientes, el servidor escucha en un canal default, conocido por el cliente. Cuando recibe una conexión, se le asigna al cliente otro canal y se crea un thread dedicado exclusivamente a ese cliente.

En base a esto el pseudocódigo del servidor quedó así:

```
Pseudocódigo del servidor
   Server(){
      Load()
      Crear thread de partidos
      Crear thread de guardado de datos
      Crear thread de trades
      Crear thread de timeout de conexión
      Configurar canales default
      While(1){
         Escuchar en el canal default // Bloqueante
         Configurar nuevos canales de cliente
         Crear thread para atender al cliente y derivárselo
      }
   }
```

Al igual que con el cliente, una de las tareas más difíciles fue implementar el draft. Para esto, se decidió tener un thread aparte que ejecutara el draft. Inicialmente se pensó que tanto el thread del draft como el del servidor dedicado estarían escuchando al cliente. Sin embargo, como con esa disposición sería posible que el servidor le "robara" el mensaje al thread, se decidió que sólo el servidor escuchara al cliente y, si el mensaje era para el draft, el servidor se lo comunicaría mediante un pipe.

Otro problema importante encontrado fue el cómo darse cuenta que la conexión de un cliente se cayó. A este problema no se le encontró solución. Fue necesario recurrir al tutor que propuso tener un thread aparte que cada cierto tiempo viera la diferencia de mensajes enviados por el cliente antes y después de contar el tiempo. Si la diferencia era 0 se consideraba que el cliente perdió la conexión por "timeout".

Cabe destacar además que se "catchean" señales tales como ctrl+Z y ctrl+C, mas si se produce una salida anormal del programa, como lo es un envió de estas señales, será responsabilidad del usuario configurar los IPCs en su máquina para un próximo uso de la aplicación. Esto puede ser utilizando el comando `ipcrm`.

Implementación de IPCs

En cuanto a la implementación de la API de IPCs, primero se optó por utilizar dos punteros a estructuras que guardaban los datos correspondientes a cada canal, "donde recibe el cliente" y "donde recibe el server". Esta estructura es denominada Channel y varía en cada implementación de IPC.

Para FIFOs, message queue y shared memory no hubo problema, pero a la hora de implementarlo con socket se nos presentó la siguiente situación:

Cuando el cliente ejecutaba `receivePacket()` o `sendPacket()` las llamaba con:

```
receivePacket(bufer,size,clientListen/*donde escucha el cliente*/);

sendPacket(buffer,size,serverListen/*donde escucha el server*/);
```

Pero las funciones de socket `receivefrom()` y `sendto()` poseen el parametro "`&addr`", el cual modifican en su ejecución. Entonces, como `receviePacket()` involucra a un canal y `sendPacket()` involucra al otro, los cambios en la variable "`addr`" que ocurran en uno, no se verán reflejados en el otro y esto influirá negativamente en la ejecución de `sendto()` y `recivefrom()`.

Por esta razón, se decidió rediseñar la API de IPCs contando ahora solo con una estructura del tipo Channel donde tengo información sobre ambos canales. Este diseño fue elegido como el definitivo y solo se debió agregar a las funciones `sendPacket()` y `receviePacket()` el parámetro "`who`" que indica quien invoca la función.

Con respecto a **FIFOs** no tuvimos problema alguno, más que en algunas maquinas no se podía crear el FIFO por problemas de permisos pero se crean en la carpeta `/tmp/` y el problema se elimina.

En cuanto a **Message Queue**, decidimos implementar toda la comunicación con una sola cola de mensajes, la cual analiza que mensaje corresponda a cada proceso por medio de prioridades. Así, tenemos una sola cola bidireccional para todo el trabajo, pues cada canal se identifica con una prioridad.

Para Shared Memory se nos presentaron algunos problemas de condición de carrera que solucionamos con semáforos. Aquí, contamos con dos porciones de memoria compartida entre dos procesos (uno para enviar a cada lado). Así, corregimos el problema de que se sobrescriban datos que aun no se han leído, y se pierda así un mensaje, y el de si se empieza a escribir mientras el otro proceso está leyendo del canal.

```
Pseudocódigo de envío y recepción de SharedMemory

Semaphore to_read = 0;
Semaphore to_write = 1;

read(){
   bajar(to_read);
   READ
   subir(to_write);
   setear en cero to_read;
}

write(){
   bajar(to_write);
   WRITE
   subir(to_read);
   setear en cero to_write;
}
```

En cuanto a Socket decidimos utilizar los puertos (de tipo int) para identificar cada canal de transmisión.

Se nos presento el problema antes mencionado de la variable `addr` que es modificada por las funciones `recivefrom`() y `sendto`(), pero lo solucionamos con la nueva implementación.