

Sistemas Operativos

Trabajo Práctico 1

IPCs y Servidores Concurrentes

2do Cuatrimestre

2011

Integrantes

José Ignacio Galindo

Federico Homovc

Nicolás Daniel Loreti

Índice

Introducción	4
---------------------	----------

Procesos	5
-----------------	----------

Implementación IPCs	6
----------------------------	----------

FIFOs	7
--------------	----------

Sockets	8
----------------	----------

Message Queue	10
----------------------	-----------

Shared Memory	12
----------------------	-----------

Resultados	16
-------------------	-----------

Análisis de Resultados	23
-------------------------------	-----------

Conclusiones	24
---------------------	-----------

Anexos	26
---------------	-----------

Introducción

Este trabajo consiste de una aplicación que simula la distribución de medicinas entre ciudades por medio de empresas que utilizan aviones. Cada ciudad posee una necesidad inicial de medicamentos de algún tipo, y cada aerolínea tiene una cierta cantidad de aviones cada uno con uno o más medicamentos iniciales.

La aplicación está dividida en distintos procesos. El principal es llamado Map y es el que inicializa el servidor para poder comunicar todos los procesos. Es el encargado de crear el proceso IO (para la salida por pantalla) y un proceso para cada una de las compañías. Cada compañía crea, además, un thread por cada avión. Al iniciarse todos los procesos de las compañías y de IO, éstos se conectan al servidor y mandan un mensaje al mapa para avisar que está listo; una vez que todos están conectados, el mapa inicia la simulación por turnos.

Un turno empieza mostrando el mapa con las ciudades y sus respectivas necesidades. Luego, cada empresa mueve todos sus aviones una unidad de distancia hacia la ciudad que se dirige y, en caso de que arribe, se lo comunica al mapa. El mapa verifica si los aviones de cada compañía pueden descargar todas o alguna de las medicinas y le envía a IO por cada compañía un mensaje con todos los aviones que descargaron. IO muestra dichos aviones en pantalla o, en caso de que ninguno descargue, un mensaje correspondiente. Luego, cada empresa reasigna el destino a sus aviones que hubiesen arribado una nueva ciudad basándose en la carga del avión, las necesidades de la ciudad y la distancia hasta la misma.

Procesos

La sincronización de los distintos procesos se lleva a cabo mediante el pasaje de mensajes. Para lograrlo, todas las lecturas y escrituras de mensajes (mediante las funciones de C `read()` y `write()` en el caso de fifo, `sendto()` y `recvfrom()` en sockets, `msgrcv()` y `msgsnd()` en las colas de mensajes, e implementándolo de forma interna en memoria compartida) son bloqueantes, por lo que en más de una ocasión un proceso debe esperar hasta que otro haya ejecutado cierto fragmento de código y mandado un mensaje de confirmación para poder continuar.

En cuanto a los threads de los procesos de las compañías, se utilizan dos variables de tipo mutex y dos variables de condición: una de cada tipo para los aviones y una de cada tipo para la compañía. Los aviones inicialmente se crean y se bloquean esperando sobre su variable de condición, luego la compañía hace un broadcast sobre esa variable y espera sobre su variable de condición hasta ser despertada una vez por cada avión. De esta forma, se asegura que una vez despierta todos los aviones ya hayan concluido su trabajo. Este mecanismo se repite a través del código de la compañía, cada vez que los aviones realizan una de las siguientes acciones: verificar si puede descargar alguna medicina cuando llega a destino, actualizar su carga en caso de haber descargado, y buscar una nueva ciudad destino.

La variable mutex de los aviones sirve para evitar que más de un thread modifique una variable global al mismo tiempo, con lo que se perderían datos sobre qué avión llegó a una ciudad y cuál no. Cada avión debe bloquearla cuando comienza a ejecutar y desbloquearla cuando está esperando. La variable mutex de la compañía sirve para esperar a que un thread se duerma antes de que otro quiera despertarlo, con lo que se producirían dead locks y el proceso se bloquearía irremediablemente.

Implementación IPCs

Para la implementación de la capa de transporte se decidió utilizar una interfaz unificada para los distintos IPCs. Esta interfaz utiliza el concepto de cliente-servidor, que consideramos adecuada para el modelo de problema que se nos planteó, ya que en nuestro caso se pensó en el mapa como una especie de servidor que no sólo crea a todos los demás procesos, sino que además controla el único recurso compartido entre todos: el grafo del mapa, procesando las solicitudes de cada uno de los clientes. De esta manera, el mapa es el encargado de inicializar el servidor, que dependiendo del tipo de IPC, utiliza una implementación del ADT serverADT distinta. De esta forma, como map inicializa el server antes de crear a los demás procesos, todos los procesos pueden acceder a la información del server después de las llamadas a `fork()`. Aquí es conveniente mencionar que en un primer momento se había decidido utilizar llamadas a `exec()` después de hacer el `fork()` para crear a todas las empresas, pero por la cuestión de compartir la información inicial del servidor, y por considerarlo una solución más simple, se decidió utilizar solamente `fork()`.

Una vez creado un proceso, éste hace una llamada a `connectToServer()` que recibe justamente como parámetro del ADT del server, y de esa manera obtiene el ADT que identifica a su cliente, con el que le es posible comunicarse. Como se puede pensar, el mapa además de crear el servidor también tiene que conectarse al servidor a través de la función anterior. Una vez conectado, usará el pid de cada uno de sus hijos (que conoce, ya que el mismo los concibió) para así obtener cada uno de los clientes y poder comunicarse. Por ende, el mapa envía mensajes a cada uno de sus hijos (los hijos no se comunican entre si) y cada uno de los hijos se comunica con él enviándole mensajes a través del cliente del mapa, que puede conseguir, ya que como es su padre, sabe su pid.

Una vez terminada la simulación, el mapa llama a la función `terminateServer()` que se encarga de liberar la memoria reservada por el servidor y cualquier otro recurso necesario (por ejemplo archivos para fifo y sockets). A su vez, cada proceso (incluido el mapa) ejecuta la función `disconnectFromServer()` para liberar sus recursos particulares.

A continuación se explica detalladamente como implementó este mecanismo cada IPC, que en algunos casos difiere en gran medida debido a, por ejemplo, la implementación de threads internos.

FIFOs

Para implementar la interfaz de transporte utilizando Fifo se pensó en la idea básica de una conexión cliente-servidor: los clientes se deben conectar al servidor primero para luego poder mandar mensajes entre si. Siguiendo este planteo, al crearse el servidor mediante la función `createServer()`, se abre un archivo Fifo para el mismo y se inicializa un thread que constantemente espera leer algun dato escrito en dicho archivo.

Cuando un cliente se quiere conectar al servidor, mediante la función `connectToServer()` se reserva espacio para dicho cliente, se abre un archivo Fifo para lectura y escritura, y se escribe un mensaje en el Fifo del servidor con el nombre de este archivo y el pid del cliente. En este momento, el thread del servidor lee los datos escritos y crea un nuevo cliente con los mismos, luego guarda la variable cliente en un vector de clientes que posee el servidor. Este vector es necesario para la función `getClient()` mediante la cual un proceso puede obtener el cliente de otro conociendo su pid y así comunicarse. Cuando un proceso termina de comunicarse, el thread del servidor le manda un mensaje para asegurarse de que el cliente se ha creado satisfactoriamente. Este mensaje también sirve para sincronización ya que de lo contrario la función `connectToServer()` podría retornar antes de que el thread ubique el cliente en el vector, con lo que un proceso podría estar buscando en el servidor un cliente creado pero que aún no está guardado en el mismo y esto inevitablemente produciría un error.

Para enviar y recibir mensajes se utilizan las funciones `sendMsg()` y `rcvMsg()` que internamente utilizan las primitivas de C `write()` y `read()` respectivamente. Cada proceso escribe mensajes en el archivo fifo del destinatario y los lee en el propio.

La función `disconnectFromServer()` simplemente libera el espacio reservado para el cliente, mientras que `terminateServer()` cierra el archivo Fifo del servidor, luego recorre el vector de clientes cerrando sus correspondientes archivos y liberando memoria, y finalmente libera la memoria del vector y del servidor.

Con esta implementación, la comunicación entre procesos resulta extremadamente sencilla, si bien el proceso de conectar clientes a través del servidor puede ser un poco más complicado.


Sockets

A la hora de implementar sockets, dado que nuestra API es del estilo Cliente-Servidor intentamos darle un rol central a este último e intentar que todos los mensajes pasaran por el servidor o que tengan centralizada toda la información de los clientes conectados para respetar el modelo.

Para realizar esto, en primera instancia nos decidimos por usar sockets de Dominio INET y de tipo SOCK_STREAM pero la realidad es que se volvía muy engorroso usar este sistema orientado a conexión y no tenía mucho sentido implementarlo de esta forma ya que los procesos de nuestro programa iban correr solo en un ordenador sin necesidad de comunicarse a través de una red con diferentes computadoras.

Por esto último decidimos usar sockets de dominio UNIX y de tipo SOCK_DGRAM que utilizan UDP como protocolo. Como ventaja podemos decir que su implementación al no ser orientada a conexión es mucho más sencilla. Evitamos hacer llamadas a listen, connect y accept como así también llamadas para crear threads. También es mucho más liviano que un protocolo como TCP y aunque no asegura que los paquetes sean recibidos en teoría se gana velocidad (Hubiese sido conveniente poder tener ambas implementaciones y realizar comparativas para ver cuanto afectaba a la comunicación un proceso y otro). Como principal desventaja podemos mencionar que al no ser orientado a conexión no se garantiza que los paquetes lleguen sumado y que no funciona bien en algunos sistemas como Mac.

Una vez decidido el dominio y el tipo de protocolo pasamos a implementar las funciones de la API y nos pareció que lo mejor era salirse un poco de este estilo Cliente-Servidor y pasar a un Peer-to-Peer haciendo que todos los clientes esten conectados entre si, sin necesidad de pasar por el servidor. Esta decisión estuvo basada en lo fácil que es comunicarse con otro proceso usando el sistema de archivos/paths que ofrece el sistema de comunicación no orientado a conexión. Es por esto que al poner una ruta generica `"/tmp/socket_"` más el id del proceso podíamos comunicarnos con cualquiera armando y bindeando esa dirección.



Entonces, la función `createServer()` de nuestra API paso a tener un rol secundario creando un socket y luego haciendo el correspondiente binding y también creando un semáforo para su posterior utilización. `ConnectToServer()` crea un `clientADT` haciendo una llamada a socket y luego a bind usando como parámetro el path formado `"/tmp/socket_"` y el id del proceso. `getClient()` Se encarga de devolver un `clientADT` apto para poder usarlo en una comunicación entre procesos basándose en el id que recibe como parámetro.

Y luego `sendMsg()` y `recvMsg()` que hacen los correspondientes llamados a `sendTo` y `receiveFrom` que son necesarios en este tipo de comunicación (No se usan `send()` y `receive` como en TCP).


Para finalizar tenemos las funciones para desconectar clientes y el servidor que solamente se encargan de hacer los unlinks y free's correspondientes.

Message Queue

La implementación de colas de mensajes resultó ser la más simple en términos de complejidad y longitud del código. Se decidió implementar el ADT del servidor con un único entero llamado `queueID`, que como su nombre lo indica, guarda el identificador de la única cola de mensajes creada para que todos los procesos (clientes) puedan comunicarse. En este punto es necesario mencionar que se optó por usar una sola cola de mensajes, sin prioridades (ya que no se consideraron necesarias para el problema planteado), donde cada "casillero" de la cola de mensajes esté identificado con el pid del proceso comunicado. De esta manera un proceso recibe mensajes desde su cliente (que justamente está implementado sólo con un entero `queueID` y su pid), es decir que escucha en su pid y envía mensajes a otros procesos haciendo un `msgsend()` con el pid del mensaje destinatario. De este manera, la implementación de colas de mensajes se adaptó de forma idónea al modelo del proyecto, ya que al contar el mapa con los pid de todos sus hijos, simplemente envía mensajes usando `msgsend()`, utilizando como parámetro el pid de cada hijo; y cada hijo, al conocer el pid del padre, es decir del mapa, envía mensajes al mismo haciendo `msgsend()` con su pid. Por ende, en cada momento de tiempo en cada "casillero" de la cola de mensajes de cada hijo hay un solo mensaje, pero en el "casillero" del mapa, los mensajes se van encolando, llegando a haber en la cola tantos mensajes como aerolíneas hubiese.

En cuanto a la creación del servidor, representado como se mencionó anteriormente por `serverADT`, se realiza dentro de la función `createServer()`, y básicamente se obtiene la identificación de la cola(`queueid`), usando la función `msgget()`, utilizando el flag `IPC_CREAT` – para que sólo se cree la cola si no existe previamente- y una clave (`queueKey`), que consta de un número elegido arbitrariamente.

Para conectarse al servidor, se utiliza la función `connectToServer()`, que simplemente, crea el espacio para un nuevo cliente, representado por `clientADT`, copia el valor del `queueID` y luego inicializa el pid del cliente con su pid. El funcionamiento de `getClient()` es similar al de `connectToServer()`, con la diferencia que en vez de inicializar el pid del ADT con el suyo, lo hace con el que le es pasado como parámetro.



Para el envío y la recepción de mensajes se usa otra estructura denominada `msgQueue` que, contiene un `long mtype`, que guarda el `pid` del mensaje enviado (para indicarle a la cola de mensajes que el mensaje se debe guardar en la "ranura" dado por el valor de `pid`), y un arreglo de `chars` llamado `mtext` que es inicializado con un tamaño prefijado dado por la constante `MSG_SIZE`.

En las funciones `disconnectFromServer()` y `terminateServer()`, sólo se destruyen las estructuras del cliente y del servidor respectivamente, haciendo un `free()` de los mismos. También en `terminateServer()` se destruye la cola de mensajes haciendo una llamada a la función `msgctl()`.

Shared Memory

En el caso de memoria compartida, se pensó en un momento en hacer una sola llamada `shmget()`, de manera de obtener un solo bloque que memoria, que a su vez iba a ser dividido en varios bloques, uno por cada cliente, partiendo desde una cantidad de clientes máxima prefijada. Cada uno de estos bloques iba a estar compuesto por una lista circular de mensajes, y de esta manera la implementación del ADT del cliente iba a tener un puntero a void para apuntar a la memoria compartida, un entero usado como offset (para indicar el comienzo de su bloque) y dos enteros adicionales para indicar el comienzo y el fin de la cola dentro del bloque. De esta manera, se suponía que un si un cliente se quería comunicar con otro iba a escribir un mensaje en la lista circular dentro del bloque el mensaje destinatario, y para recibir un mensaje un cliente tendría que leer de la lista circular de su propio bloque. Así podrían convivir varios mensajes en un mismo momento para cada cliente, siempre manteniendo un tamaño fijo para cada mensaje.

Finamente se optó por descartar la implementación anterior, encontrando una forma considerablemente más simple de resolver el problema presentado en el diseño del IPC. Se decidió descartar las listas circulares y, en vez que hacer un sola llamada a `shmget()`, hacer dos, de manera de obtener dos bloques de memoria compartida diferentes; uno que se usaría para los mensajes en sí, y otro que se usaría para guardar la información de los clientes conectados al servidor. De esta manera, una memoria compartida quedaría integrada por un arreglo de estructuras "shmMessage", formadas principalmente por arreglos de caracteres de un tamaño predefinido para almacenar los mensajes, y otra que quedaría conformada por un arreglo de estructuras `clientCDT` y que guardaría la información de cada cliente conectado al servidor.

Para concretar este diseño se decidió implementar el tipo abstracto `serverADT` de la siguiente manera:

```
struct serverCDT {  
    int semid;  
    int shmidClients;  
    int shmidMessages;  
    void * clients;  
    void * memory; };
```


Con esta estructura, el server puede guardar toda la información necesaria para lograr de forma efectiva la comunicación entre distintos clientes. El campo `semid`, se usa para guardar el identificador de un arreglo de 2 semáforos system V (implementados en el archivo `semaphore.c`), los cuales se usan para garantizar la exclusión mutua entre clientes en las memorias compartidas. De forma análoga, los campos `shmidClients` y `shmidMessages`, contienen los dos identificadores de las memorias compartidas utilizadas, de manera que cada cliente (proceso) pueda vincularse con la memoria compartida. En cuanto a los punteros a void restantes, se utilizan para que el servidor se vincule a las memorias compartidas, pudiendo así inicializarlas en 0, mediante la función `cleanUP()`, tarea que se realiza al final de la función `createServer()`.

En lo que se refiere a la implementación del ADT de `clientADT`, fue realizada de la siguiente manera:

```
struct clientCDT {  
    pid_t id;  
    int used;  
    int semid;  
    int shmidMessages;  
    int offset;  
    void * memory;  
};
```

El campo `id` guarda el pid del proceso presente en la comunicación y es la forma de asociarlo con el cliente que le corresponde. El campo `used`, como su nombre lo indica, se utiliza para señalar si el cliente en cuestión está siendo utilizado por un proceso o no.


Este campo no se utiliza para la comunicación en sí, sino que lo utiliza el servidor para saber si una determinada posición dentro del vector de memoria compartida de clientes ya se encuentra tomada por un proceso. El campo `semid`, así como en el servidor, contiene el identificador del arreglo de dos semáforos que se utilizarán para garantizar la exclusión mutua cuando se trata de escribir un mensaje en la zona de memoria compartida destinada para dicha finalidad.



Cabe destacar que cuando se usa este campo, dentro de las funciones `sendMessage()` y `rcvMessage()`, se utilizarán las funciones de incremento y de decremento del semáforo, con el argumento `SEM_MEMORY`, para indicar que se usa el semáforo destinado para la exclusión mutua correspondiente a la memoria compartida de mensajes (un semáforo del arreglo sirve para cada memoria). Los campos restantes son usados por las funciones de envío y recepción de mensajes para escribir en el espacio destinado para los mensajes del cliente utilizando los mensajes correspondiente, valiéndose de offset para este objetivo

En forma general, la implementación en cuestión funciona de la siguiente manera: al llamar a la función `createServer()` se crea una nueva instancia del tipo abstracto del servidor, obteniendo a partir de las claves destinadas a este uso los identificadores de tanto los semáforos como las memorias compartidas que se utilizarán para la comunicación. Luego se obtienen los punteros a dichas memorias y se las inicializa a todas en cero llamando a la función `cleanUp()`. Una vez obtenido el `serverADT` cada uno de los procesos involucrados llamará a la función `connectToServer()`, para así obtener el `clientADT` a través del cual podrá comunicarse. Esta función utilizará el semáforo destinado para la memoria compartida correspondiente al arreglo de clientes para asegurarse que no hayan problemas de exclusión mutua, buscando dentro del arreglo de clientes en la memoria compartida el primer cliente no inicializado. Una vez ubicado el cliente, y si la cantidad máxima de clientes lo permite, inicializará el cliente en cuestión en la memoria compartida con los datos del proceso y luego, creando un nuevo `clientADT` para devolverlo como valor de retorno, copiará en él los datos del cliente recién inicializado en la memoria compartida. Al inicializar el cliente, también colocará su campo offset apuntando al espacio de memoria compartida de mensajes que le corresponde.

Una vez que el proceso ya tiene su `clientADT`, es decir que se encuentra conectado al servidor, deberá hacer una llamada a `getClient()`, para así poder comunicarse con otro proceso, simplemente indicando su pid. De esta forma, en la función `getClient()` se buscará en la memoria compartida para los clientes, el cliente cuyo pid coincida con el indicado el cliente, y si lo encuentra, inicializará un nuevo `clientADT`, copiará los datos y devolverá. Teniendo así el cliente con el cual puede comunicarse con otro proceso, llamará a la función `sendMessage()` que simplemente, recibiendo la información del cliente



destinatario, buscará en el arreglo de memoria compartida para el uso de los mensajes, el lugar donde se encuentre el espacio reservado para los mensajes de dicho cliente, y así, utilizando memcpy, guardará en mensaje enviado, colocando en el flag isWritten en true.

Aquí es importante aclarar que este IPC, a diferencia de los demás, no puede guardar más de un mensaje para un mismo cliente simultáneamente. La manera en que se soluciona este problema es la siguiente: cuando el flag isWritten de la estructura shm-Message correspondiente a un determinado proceso se encuentra activo, el proceso que envía el segundo mensaje, se bloquea en un ciclo hasta que el mensaje en dicha posición sea leído y así el flag sea desactivado. De esta manera se logra implementar este ipc de una manera simple, no impidiendo el correcto funcionamiento del mismo.

Resultados

Todas las mediciones fueron realizadas dentro de una máquina virtual (Virtual Box) corriendo Ubuntu 11,4 32 bits, utilizando como sistema operativo anfitrión Mac OSX Snow Leopard (10.6.8) y usando como equipo una computadora MacBook Pro , modelo 8,4 , con un procesador Intel Core i7 2Ghz de cuatro núcleos. Todos los resultados están medidos en milisegundos y se realizaron 5 simulaciones por testeo para tener un menor margen de error.

Tipos de Mediciones

Aviones y Compañías Fijos

Se usaron varios mapas los cuales poseían un tamaño fijo de 10 aviones y 10 compañías. La idea de dejar fijos estos dos parametros es ver como se desenvuelven las distintas implementaciones de los distintos IPCs (Shared Memory, FIFOs, Sockets y Message Queue) conforme van cambiando la cantidad de ciudades.

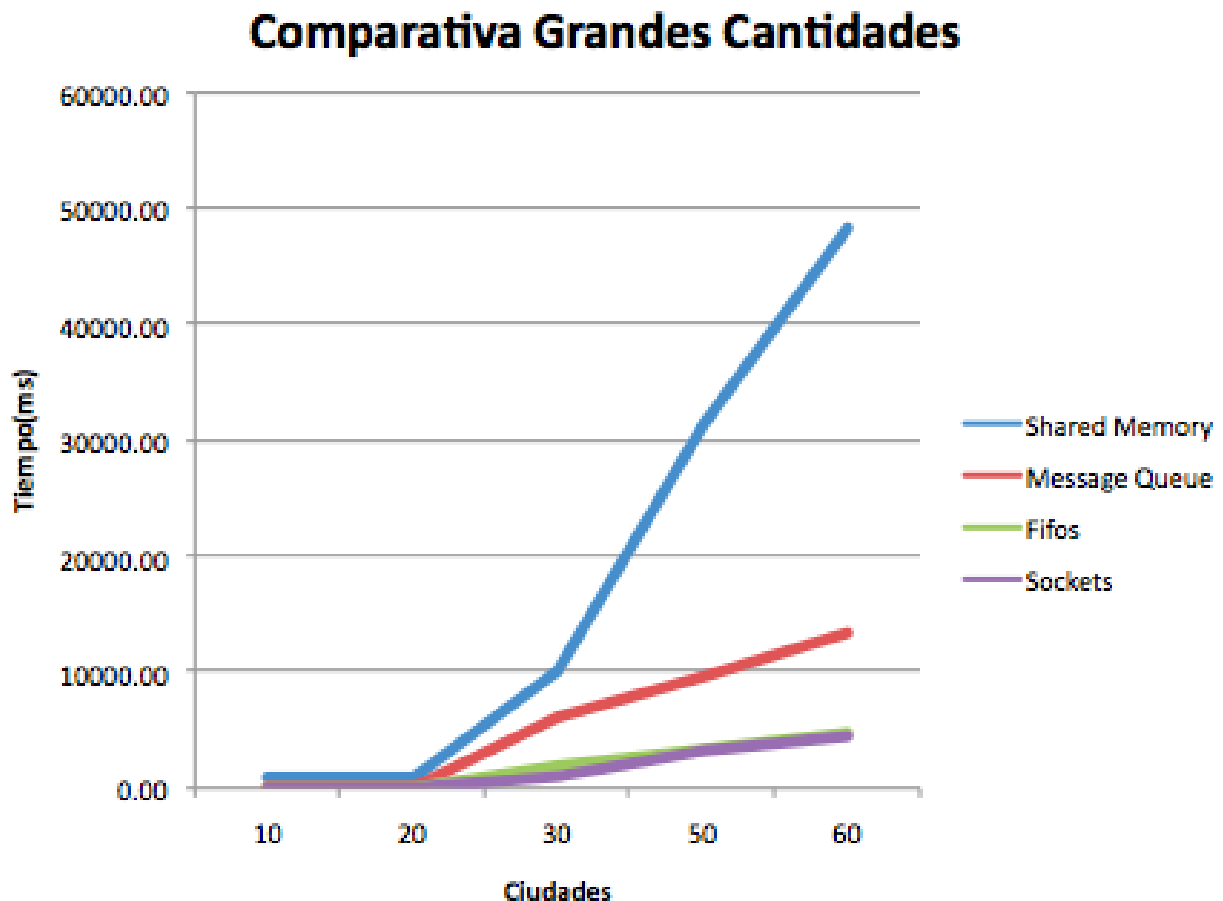
También realizamos una comparación entre FIFOs y Sockets ya que veíamos desde el punto de vista del código y de la implementación que eran sumamente parecidos.

Compañías y Aviones Variables

Se usaron tres mapas diferentes para hacer estos testeos. Todos con 30 ciudades cada uno y el mismo grafo.

La idea consistió en ir variando la cantidad de aviones y compañías en los distintos niveles e ir analizando su performance a través del tiempo requerido en terminar una simulación y distribuir con éxito la totalidad de los suministros requeridos por las diferentes ciudades.

Aviones y Compañías fijos

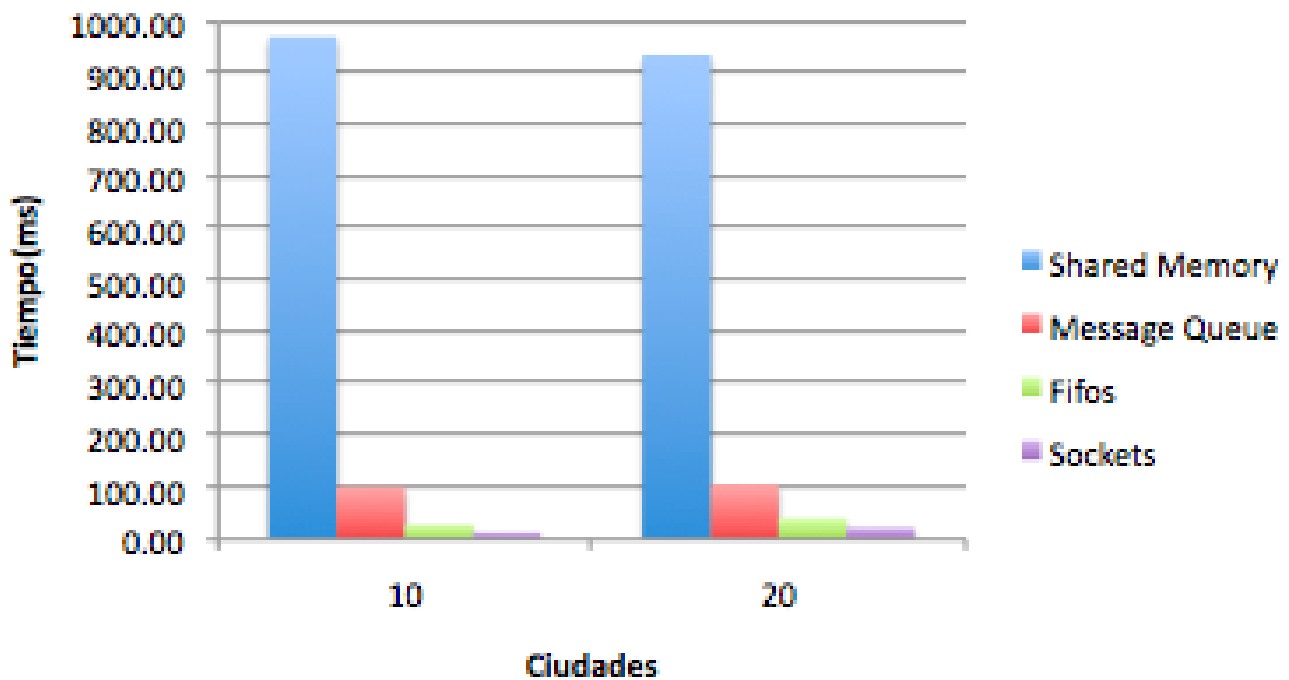


En este gráfico podemos ver como para cantidades chicas de aviones no se aprecian muchas diferencias de tiempo considerables para resolver los mapas. Sin embargo, cuando nos vamos a mayores cantidades se empieza a ver claramente cuál es el rendimiento de cada IPC implementado.

Podemos observar como Shared Memory se vuelve mucho más lento que los otros IPCs en el momento que superamos la barrera de las 30 ciudades. Message Queue tiene un rendimiento aceptable y Socket y Fifo se ubican como las implementaciones más rápidas, prácticamente sin diferencias notables.

(Más adelante analizaremos esta paridad entre Fifos y Sockets)

Comparativa Pocas Cantidades

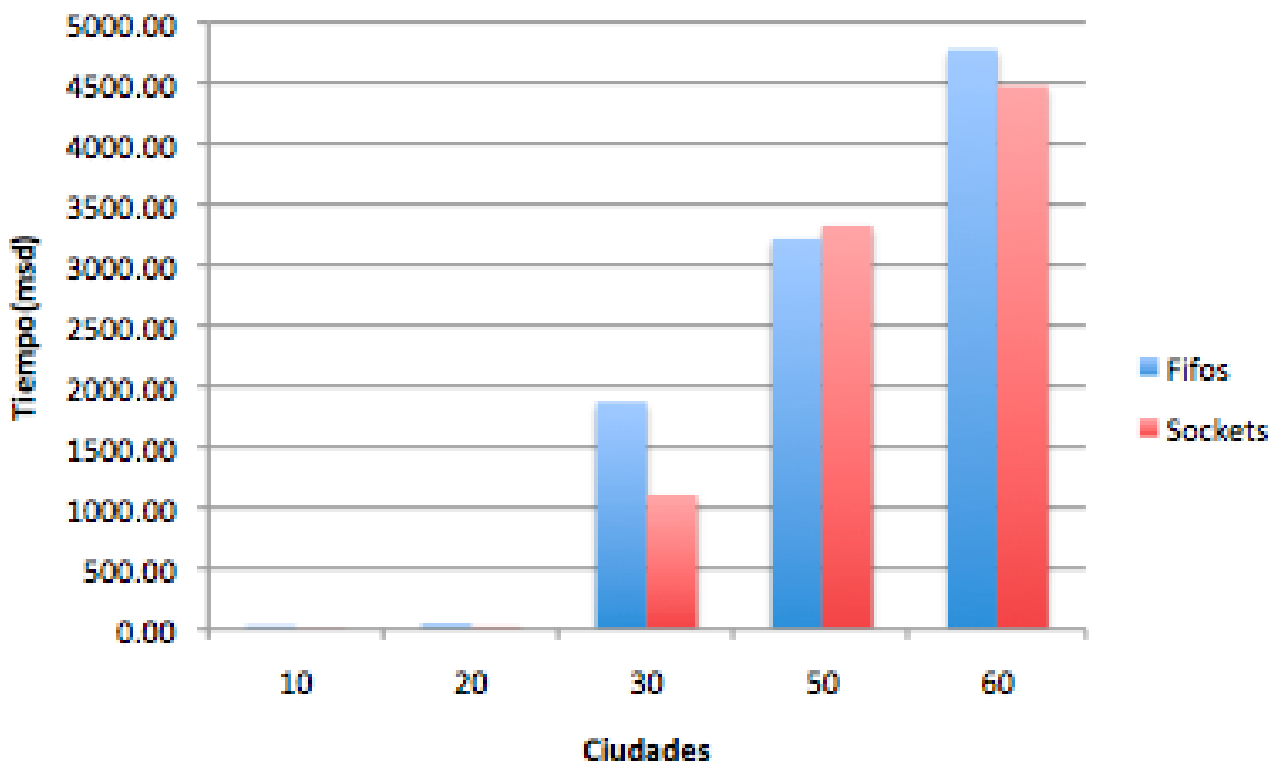


En el gráfico anterior no se podía apreciar para pocas ciudades los rendimientos de los distintos IPCs ya que la escala no lo permitía. Enfocándonos solamente en 10 y 20 ciudades, podemos ver como Shared Memory sigue siendo el más lento de las 4 implementaciones y como no se notan significativas diferencias entre Message Queue, Fifos y Sockets.

La diferencia entre los tres últimos mencionados anteriormente y Shared Memory es notable. Esta última, tarda casi 10 veces más que el resto de las implementaciones en terminar la simulación.

Tres de los cuatro IPCs resuelven el mapa en menos de 100 milisegundos mostrando un muy buen rendimiento, mejor de lo que se podía esperar.

Rendimientos Socket vs Fifos

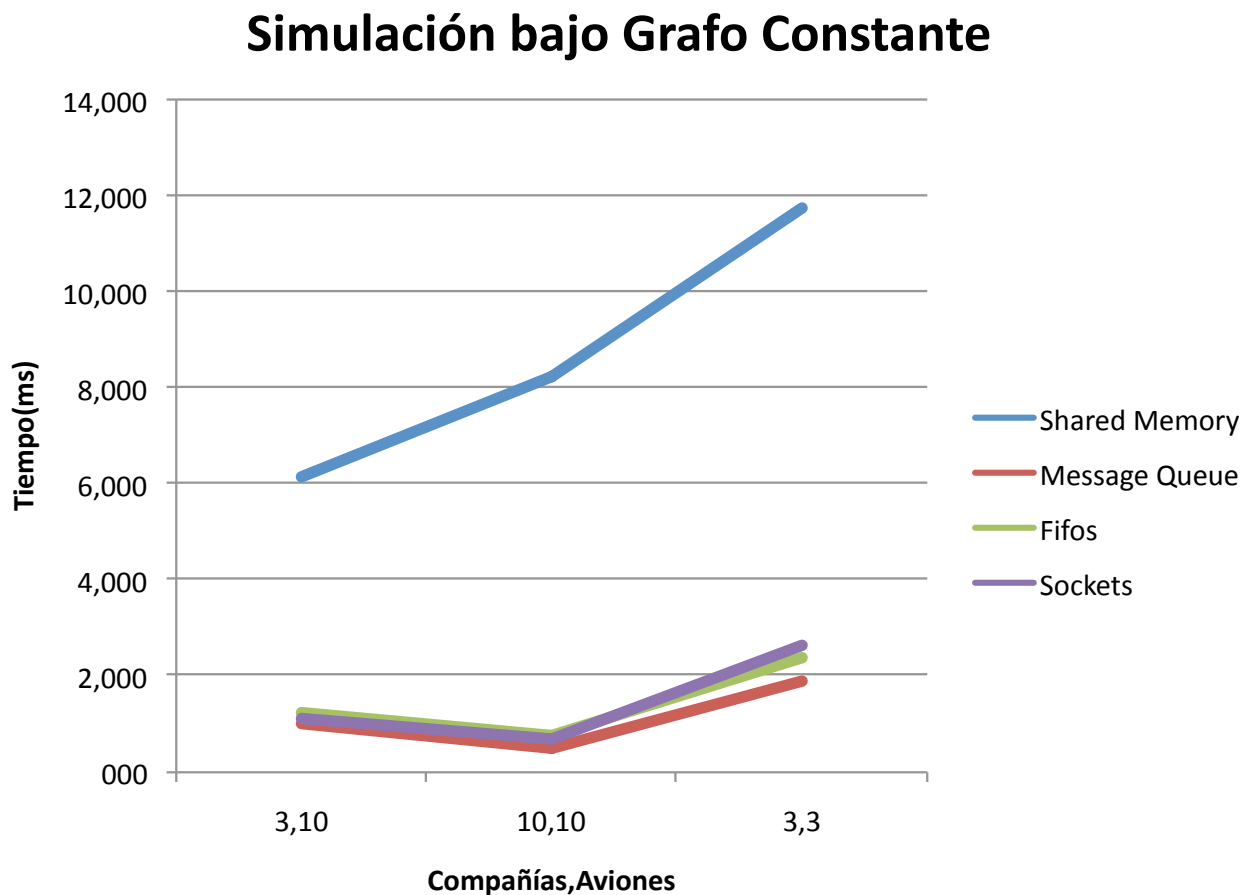


Como hemos visto en los gráficos anteriores, tanto la implementación con FIFOs como Sockets no presenta diferencias significativas conforme vamos variando la cantidad de ciudades en la simulación.

Por esto último, decidimos hacer una comparativa especial y ver como se comportaban estos dos IPCs más detalladamente (Para 10 y 20 es prácticamente nulo y no se puede apreciar en el gráfico).

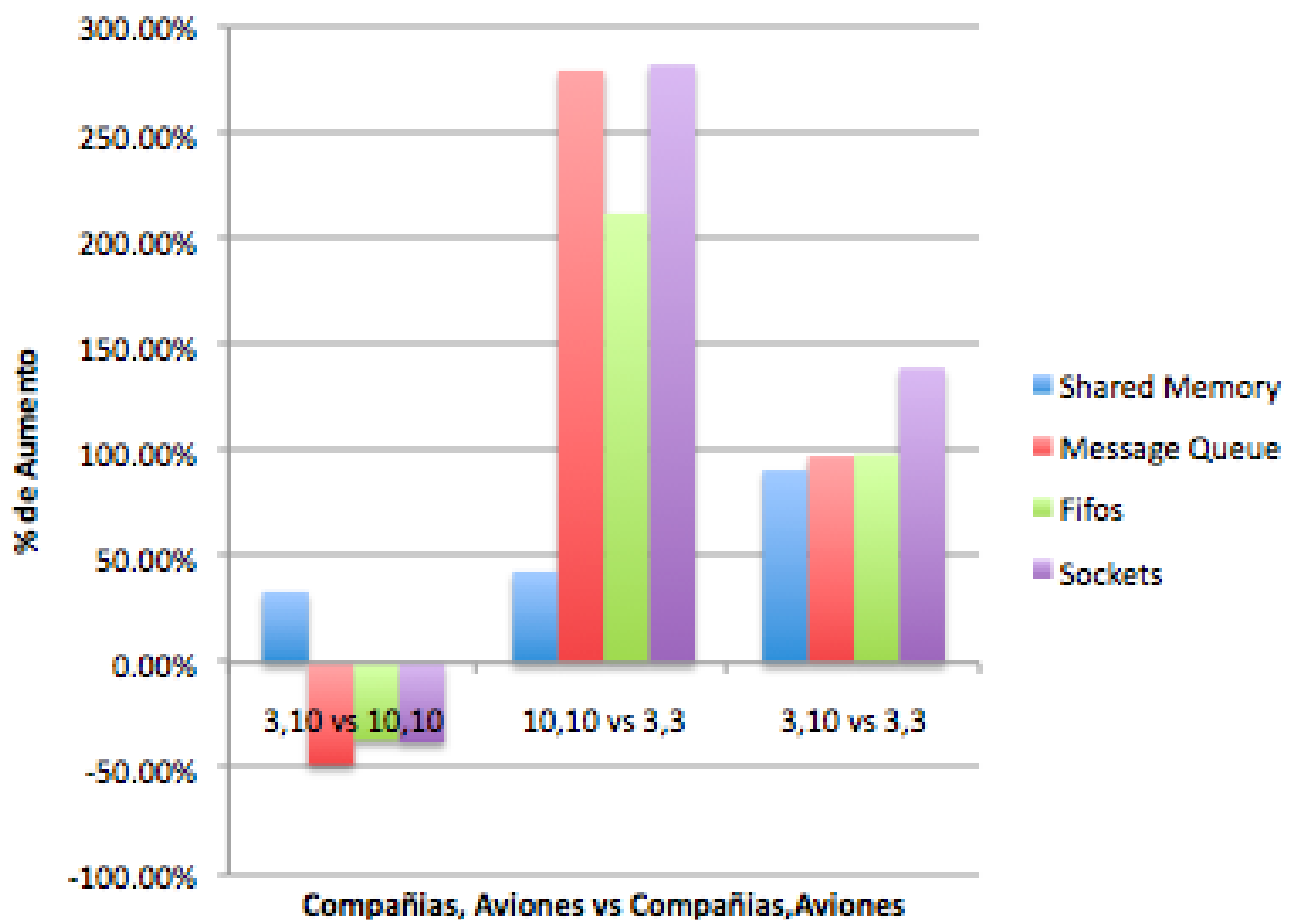
Se puede ver como Socket es un poco más eficiente que FIFO, pero en definitiva es imperceptible para el usuario. Esta paridad se debe a que la implementación de estos dos IPCs es muy parecida. Ambas trabajan con archivos y no hacen grandes manejos en memoria ni se apoyan en el uso de múltiples semáforos para poder funcionar correctamente.

Aviones y Compañías Variables



En este caso al mantener la cantidad de ciudades y el mapa constante las diferencias entre Shared Memory y los otros tres IPCs se hacen más evidentes.

También podemos ver que Message Queue tiene un rendimiento mucho más aceptable y que incluso obtiene tiempos inferiores a FIFO y Sockets que habían sido los de mejor desempeño en la antigua configuración en la que dejábamos los aviones y compañías fijos y las ciudades variables.

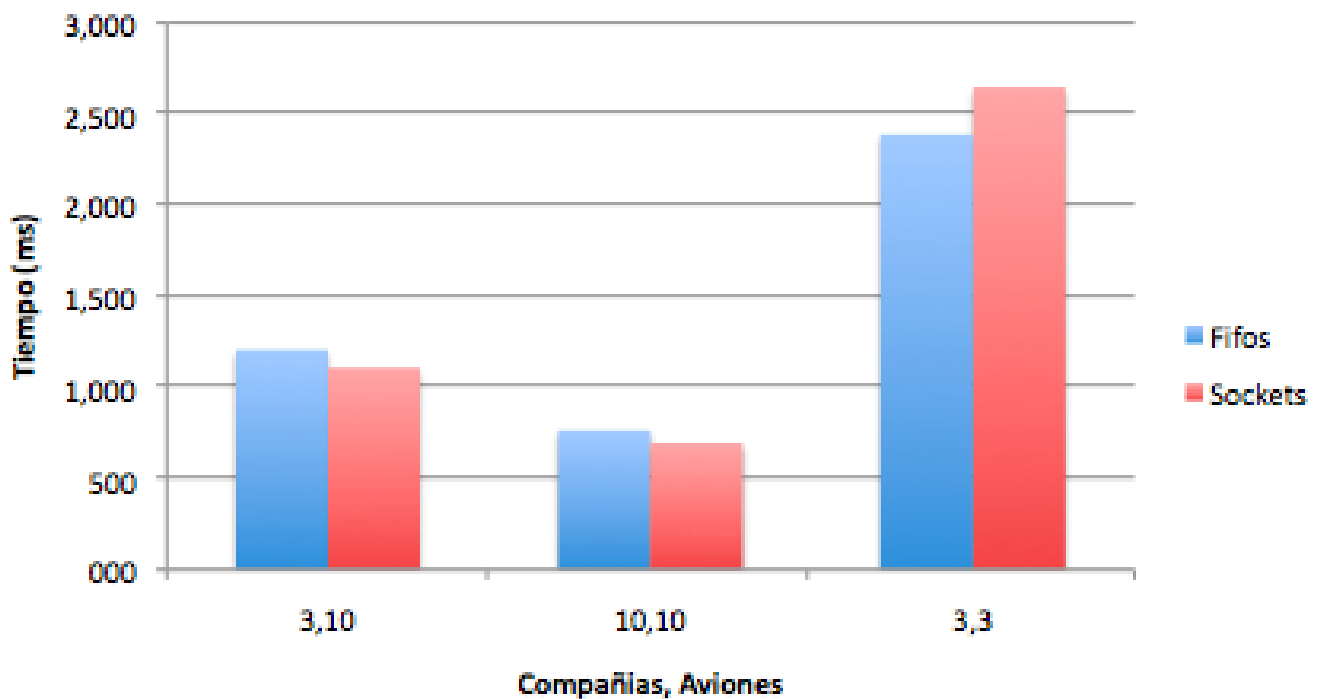


En este gráfico podemos ver cuál es la tasa de crecimiento o decrecimiento de una configuración frente a otra. En primer caso, al comparar 3 compañías y 10 aviones vs 10 compañías y 10 aviones todos los IPCs salvo Shared Memory obtienen mejoras en sus tiempos de simulación.

En el segundo caso se compara 10 Compañías y 10 Aviones contra 3 Compañías y 3 aviones. Sorpesivamente, los IPCs que esta este momento venían siendo más eficientes sufren grandes aumentos en comparación con Shared Memory.

En el tercer y último, todos se comparten de una menra similar y tienen aumentos que rondan en el valor del 100%, es decir que duplican la duración de la simulación.

FIFO vs Socket



En esta nueva configuración volvimos a comparar dos IPCs que habíamos visto que se comportaban de manera similar y volvimos a obtener resultados muy similares.

Es decir que no importa mucho que parametros cambiemos en cuanto al mapa, la cantidad de ciudades, las aerolineas o compañías y los aviones que estas dos implementaciones se van a comportar de manera muy similar obteniendo prácticamente los mismos resultados a la hora de hablar de rendimientos y tiempos de simulación.

Análisis de Resultados

Como se puede apreciar en los resultados, Shared Memory, paradójicamente a lo que se pensaba que iba a suceder, terminó siendo el ipc más lento, lo que consideramos debe estar relacionado al manejo de memoria que se utiliza, donde para agregar un cliente, como para encontrarlo se debe recorrer gran parte de la misma.

También pudimos encontrar y apreciar un gran parecido entre lo que son los rendimientos e implementaciones de FIFO y Sockets. Pudimos ver como se comportan de igual manera y responden de manera similar a los diferentes cambios o escenarios que se les presenten, sin tener un peso determinante las variables del programa (cantidad de ciudades, aerolíneas y aviones).

Mediante el anteúltimo gráfico nos pudimos dar cuenta de varias cosas, entre ellas que el rendimiento de 3 de los IPCs (Message Queue, FIFO y Sockets) mejoraba notablemente cuando incrementábamos la cantidad de aerolíneas que intervenían en la simulación. Esto se debe a que al aumentar la cantidad de aerolíneas (procesos) y mantener fija la cantidad de aviones (threads) hay más "disponibilidad" de canales para comunicarse por cada uno de los aviones. Esto tiene un efecto negativo para Shared Memory ya que al haber más aerolíneas, tiene que manejar más bloques de memoria.

Por otro lado, en ese mismo gráfico podemos observar como dejando fija la cantidad de aerolíneas (procesos) y variando la cantidad de threads repercute en cada uno de los IPCs de una manera muy similar. La caída en el rendimiento de cada uno de ellos al reducir la cantidad de aviones(threads) de 10 a 3 produce un impacto negativo próximo al 100% en los 4 casos.

Mediante el análisis de todos los gráficos podemos concluir que tanto FIFOs como Sockets son las alternativas más estables para usar en cualquier tipo de situación y poder obtener así los mejores resultados si hablamos en terminos de performance.


Conclusiones

Ciertamente este trabajo fue uno de los que trajo más dificultades en lo que todos los integrantes del grupo llevamos de carrera. Ocurrencias como comentar una variable que no era usada o cambiar de lugar un `sleep()` en la fase de prueba, podían ocasionar el fin del funcionamiento de la aplicación. Se tuvieron también muchos problemas en las primeras etapas del desarrollo, ya que uno se encontraba en frente de un trabajo cuya dificultad sobrepasaba lo acostumbrado, sin una idea clara de por donde empezar. Con el paso de las clases teóricas, después de la reiterada lectura del libro “Unix System Programming”, de la detallada inspección de las implementaciones de los ipc's publicados en iol, y de las consultas a tps anteriores, logramos enviar y recibir los primeros mensajes con `msgqueue` y `fifos`.

Paralelamente, íbamos desarrollando lo relativo a la capa de aplicación, no sin tener nuestras dificultades. En un primer momento tuvimos muchas dudas de que manera implementar el mecanismo de turnos planteado en el enunciado. Pensamos en hacer un mecanismo sin turnos, o mejor dicho, donde cada “turno” correspondía con la llegada de un avión a destino, donde cada aerolínea competía en llegar sus destinos. No tardamos mucho tiempo en darnos cuenta que esta manera era errónea, ya que nada impedía que un proceso corriese más rápido que otro, no respetando así las distancias dadas por los datos iniciales. Luego, pensamos en implementar los turnos utilizando un tiempo prefijado para cada uno, lo que también terminamos descartando por considerarlo innecesario e ineficiente. Siguiendo este camino de pensamiento, terminamos llegando a la conclusión que la mejor forma de implementar el mecanismo de turnos es haciendo que un proceso, el mapa, el padre todos los demás, funcione como “spooler”, haciendo un “broadcast” a todas compañía, y así delimitando un turno.

Volviendo a los IPCs; cuando ya teníamos las primeras ideas sobre el funcionamiento de la aplicación en general, teníamos 2 IPCs funcionando (como mencionamos antes, `fifos` y `message queues`). Al probarla con `msgqueue` la aplicación funcionó correctamente en la primera oportunidad, pero no sucedió lo mismo con `fifo`.

Antes de hacer andar `fifos`, optamos por seguir perfeccionando el funcionamiento de la capa de aplicación y empezar con la implementación de `shared memory`; `ipc`



que sabíamos iba a ser el más difícil de hacer funcionar. En este punto es donde empezamos tratando de implementar memoria compartida de la manera que se especifica en la explicación de dicho IPC. Como era de esperar, tuvimos muchos inconvenientes, sobre todo teniendo en cuenta cuestiones de condiciones de carrera y de semáforos. Finalmente de eligió cambiar la implementación a una mucho más simple, que terminó funcionando de forma correcta. Teniendo andando msgqueue y shared memory, ya nos encontrábamos en posición correcta para probar el sistema como conjunto de forma correcta.

En este momento es donde nos surgió la duda de cómo implementar el uso de “threads”, que hasta el momento no habíamos utilizado en ningún momento, ya que fifos estaba siendo replanteado desde cero. Así , surgió la idea de usar un “thread” por cada avión dentro de una compañía, utilizando como sincronización un conjunto de mutexes y variables de condición de una manera que fue explicada por los chicos de la práctica, que nos dieron un ayuda remarcable en este punto.

Habiendo concretado y testeado lo anterior, retomamos la implementación de los otros dos IPCs, terminándolos justo a tiempo como para probarlos y asegurarnos de que todo funcione correctamente.

A pesar de todas las dificultades que este trabajo nos trajo, queremos remarcar el gran aporte que hizo a nuestro conocimiento y experiencia, ya que trabajar con varios procesos independientes fue un salto muy considerable con respecto a otros proyectos no paralelizables, lo que convirtió este trabajo en un experiencia muy enriquecedora.

Anexos

La siguientes tablas muestran los datos usados para la realización de los gráficos presentados en la sección de resultados:

Ciudades/IPC	Shared Memory	Message Queue	Fifos	Sockets
10	971,86	102,34	28,33	13,29
20	936,29	106,57	40,54	22,61
30	10287,82	6336,00	1871,17	1103,51
50	31544,55	9787,67	3214,51	3324,43
60	48337,34	13402,67	4776,69	4475,46

(Cias,Aviones)/IPC	Shared Memory	Message Queue	Fifos	Sockets
3,10	6148,10	961,41	1203,33	1106,28
10,10	8208,33	500,57	765,15	691,96
3,3	11743,09	1898,27	2385,56	2643,16

	Shared Memory	Message Queue	Fifos	Sockets
3,10 vs 10,10	33,51%	-47,93%	-36,41%	-37,45%
10,10 vs 3,3	43,06%	279,22%	211,78%	281,98%
3,10 vs 3,3	91,00%	97,45%	98,25%	138,92%