Un sistema IPC para el desarrollo de aplicaciones cliente–servidor en Internet[1]

Carlos Nava – Leandro León (carlos|lrleon@cemisid.ing.ula.ve)
Centro de Estudios de Microelectrónica y Sistemas Distribuidos (CEMISID)
Centro Nacional de Cálculo Científico (CECALCULA)
Universidad de Los Andes – Mérida – Venezuela

Resumen

Este artículo presenta un sistema de comunicación remota, confiable, sobre Internet, para aplicaciones cliente-servidor. El sistema implanta un protocolo con una semántica exactamente una vez, que garantiza confiabilidad en la entrega de mensajes y soporta caídas y reencarnaciones del cliente y del servidor. El protocolo se implantó sobre la capa de transporte UDP. El sistema consta de un servidor de comunicaciones por sitio y una biblioteca a encadenar a los procesos cliente y servidor. El sistema exporta una interfaz independiente de localidad, de fácil uso y utilizable por procesos multi-hilos. Pruebas de rendimiento que comparan al sistema con TCP, evidencian la eficiencia para un intercambio simple de mensajes solicitud–repuesta.

Palabras claves: Sistemas distribuidos, RPC, IPC, Comunicación entre procesos, Aplicaciones cliente-servidor.

1. Introducción

Las redes TCP/IP ofrecen dos protocolos de transporte: TCP y UDP [2]. Estos protocolos presentan inconvenientes para el desarrollo de aplicaciones cliente–servidor donde se requiere un simple intercambio de mensajes solicitud–respuesta. UDP es un protocolo no orientado a la conexión, pero no es confiable; es decir, no garantiza que los mensajes lleguen a su destino. Por su parte, TCP es un protocolo orientado a la conexión y confiable, pero la conexión imparte costes innecesarios a las aplicaciones de intercambio simple de mensajes solicitud – respuesta.

El uso de TCP involucra distintas fases: establecimiento de conexión, transferencia de datos y finalización conexión. Por sus características, aplicaciones tales como telnet o FTP utilizan de manera conveniente TCP.

Sin embargo, aplicaciones cliente-servidor que requieren un servicio simple de comunicación deben exhibir las siguientes características:

- **Ausencia de conexión:** Los costes de conexión deben ser evitados. Cuando sea posible, debe utilizarse un paquete para la solicitud y otro para la respuesta.
- **Latencia mínima:** La latencia de la comunicación debe ser reducida al tiempo de ida y vuelta de un paquete más el tiempo que tarda el servidor en procesar la solicitud.
- **Confiabilidad:** Se lidia con pérdidas y retardos de mensajes. Además, el servidor debe estar en capacidad de detectar solicitudes duplicadas y no repetir el servicio.

Con el fin de evitar los costes de conexión, una gran variedad de aplicaciones desarrollas en Internet, tales como RPC, DNS y agentes SNMP, no utilizan TCP como capa de transporte para la comunicación remota[10]. UDP por si solo no es suficiente, pues no aporta confiabilidad. En consecuencia estas aplicaciones añaden a UDP mecanismos para lidiar con las pérdidas de mensajes, mensajes duplicados, retardos, etc.

El sistema de comunicación que se expone en este artículo presenta una solución genérica para ser usado por cualquier tipo de aplicación cliente-servidor. Nuestro sistema implanta comunicación

[1] Esta investigación fue financiada por el CDCHT-ULA bajo el proyecto No. I-620-98-02-AA

remota, confiable, adecuada para aplicaciones cliente–servidor en Internet. El sistema optimiza el uso de los recursos de red y provee una semántica exactamente una vez en la entrega de los mensajes. El sistema controla fallas de comunicación, orden de entrega de los mensajes y caídas de los clientes y servidores. Esta confiabilidad es eficazmente implantada por un protocolo que maneja las pérdidas de mensajes y minimiza el número de mensajes utilizados. El sistema se realizó sobre de la capa de transporte UDP. En ausencia de fallas, este protocolo sólo utiliza dos paquetes de red para un simple intercambio solicitud – respuesta. En contraparte, TCP utiliza 10 paquetes asumiendo que se establece una nueva conexión por cada intercambio solicitud – respuesta [9].

2. Interfaz

El sistema exporta una interfaz utilizable por cualquier proceso UNIX. Los procesos pueden utilizar hilos (*threads*) POSIX. Envíos y recepciones pueden ser realizados por diferentes hilos. Cada mensaje que se envía en el sistema debe especificarse con la clase `RawMessage`:

```
class RawMsg {
      RawMsg(void* bodyaddr, unsigned bodysize,
             short int _flags   = 0,
             unsigned int  _timeout = -1);
      void setBodyAddr(void* bodyaddr);
      void setBodySize(unsigned bodysize);
      void setFlags(short int _flag);
      void setTimeout(unsigned int _timeout)
      const void* const getBodyAddr() const;
      unsigned getBodySize() const;
      short int getFlags() const;
      unsigned int getTimeout() const;
};
```

Un objeto del tipo `RawMsg` contiene la dirección del cuerpo del mensaje (datos opacos), el tamaño del mensaje, banderas de transmisión y un *timeout* que define el tiempo máximo que esta dispuesto a esperar por la respuesta a la solicitud.

A partir de este punto, consideraremos "*cliente*" al proceso que envía una solicitud y "*servidor*" al proceso que recibe la solicitud, la procesa y envía la respuesta La interfaz consiste en dos simples clases en C++:

```
class IpcClient {

      IpcClient(const IpcPort& target);
      void sendRequest(const RawMsg& msg);
      void receiveReply(RawMsg& msg);
}
class IpcServer {
      IpcServer();
      IpcPort getPort();
      const MsgId receiveRequest(RawMsg& msg);
      void sendReply(const RawMsg& msg,
                     const MsgId& msgId);
}
```

Aunque la interfaz es en C++, ella puede ser perfectamente presentada en otro lenguaje de programación.

En principio, el servidor instancia un objeto de tipo `IpcServer`. Un puerto (IpcPort) es automáticamente creado por el sistema y puede ser consultado con `getPort()`. La noción de puerto define transparentemente las coordenadas del servidor. `IpcPort` es un identificador opaco del servidor que permite independencia de localización.

Para recibir solicitudes, el servidor invoca el método `receiveRequest()`. Al invocar este método, el servidor se bloquea hasta que recibe una solicitud en `msg`, que es una instancia de `RawMsg`. `receiveRequest()` retorna un identificador único de mensaje: `MsgId`. Este identificador es utilizado posteriormente cuando se contesta la solicitud.

El método `sendReply()` es utilizado por el servidor para enviar la respuesta al cliente. `msg` contiene esa repuesta y `msgId` es el identificador de la solicitud. El identificador de mensaje permite, a otro hilo (*thread*) diferente del que recibió la solicitud, enviar la respuesta al cliente.

La clase `IpcClient` debe ser instanciada por cualquier cliente. El constructor recibe una instancia de la clase `IpcPort`. Esta acción establece el enlace (*binding*) entre el cliente y el servidor.

El cliente envía una solicitud mediante el método `sendRequest()`. El mensaje es enviado a la localidad definida por el `IpcPort` especificado en el constructor.

Luego de enviar la solicitud, el cliente invoca al método `receiveReply()`, que es utilizado para recibir la respuesta del servidor. El mensaje recibido es colocado en `msg`. `receiveReply()` puede ser invocado por otro hilo en el proceso cliente.

`receiveReply()` bloquea al hilo hasta que se recibe el mensaje de respuesta o hasta que el *timeout* definido para el mensaje expire.

La figura 1 ilustra un ejemplo de uso de la interfaz. El cliente envía la solicitud por un hilo y recibe la respuesta por otro. Del mismo modo, el servidor recibe la solicitud por un hilo y crea otro hilo para procesar la solicitud con `Servicio_Id();` posteriormente este hilo enviará la respuesta.

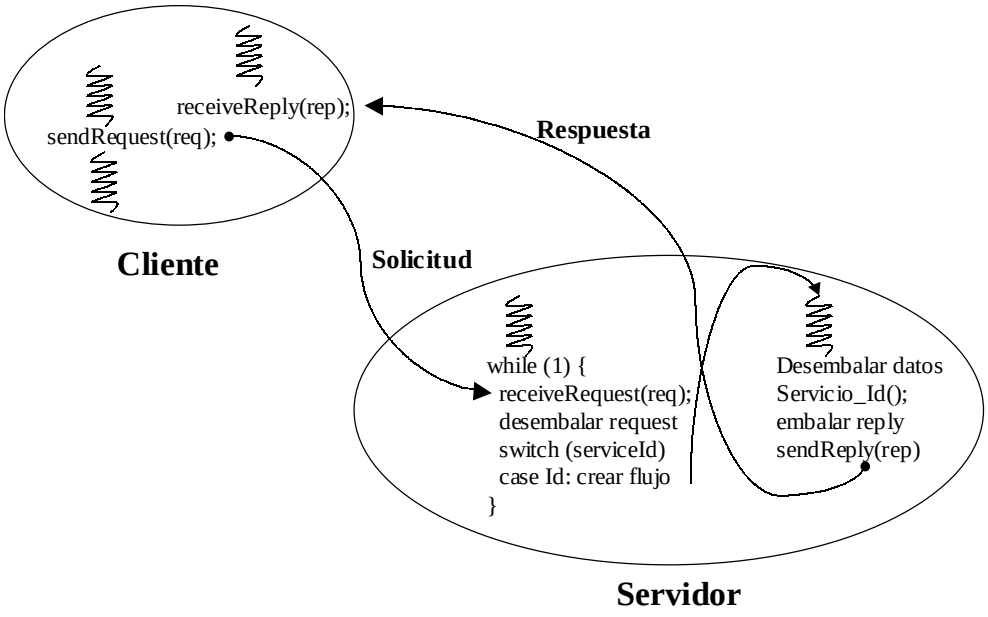

Figura 1. Uso de la interfaz del sistema

3. Protocolo

El protocolo esta basado en [7] y [8] e implanta mecanismos para lidiar con:

- Perdidas en los mensajes.
- Mensajes duplicados.
- Retardos.
- Caídas de clientes o servidores

El protocolo fue diseñado para mensajes pequeños característicos de aplicaciones cliente - servidor, RPC e invocación a objetos. La figura 2 ilustra la ubicación de nuestro sistema (capa IPC) en la estratificación por capas de TCP/IP.

Idealmente, solo se necesitan dos mensajes: *la solicitud y* la *respuesta.* La confiabilidad requiere el uso de reconocimientos, *timeouts*, retransmisiones, números de secuencia, etc. Esta confiabilidad es implantada en la capa IPC.

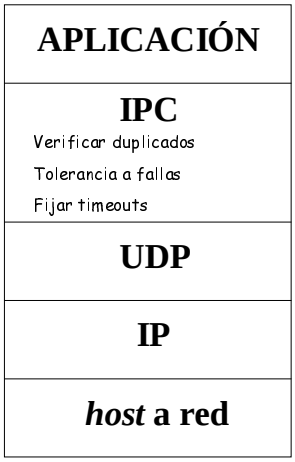

Figura 2. Estratificación por capas.

A continuación se describen los escenarios en los cuales son empleadas las técnicas mencionadas.

3.1. Pérdida de mensajes

El mensaje de respuesta funge de reconocimiento al mensaje de solicitud. Si se pierde un mensaje de solicitud, el cliente no recibe la respuesta, y por lo tanto debe retransmitir. Para ello, el cliente emplea un "*timeout*" que indica cuando debe ser reenviada la solicitud. La idea es la siguiente: el cliente envía la solicitud y espera durante un tiempo máximo (timeout). Si expira este tiempo sin recibir respuesta, entonces se reenvía la solicitud. Si el cliente reenvía la solicitud un numero de veces determinado sin recibir respuesta, entonces el cliente considera que el servidor esta fuera de servicio y aborta la solicitud.

Cuando el cliente reenvía una solicitud, él no es capaz de saber si se perdió la solicitud o la respuesta del servidor. Por lo tanto, si se pierde la respuesta, el servidor puede recibir una solicitud repetida; esta situación puede causar que el servicio se ejecute mas de una vez. Para evitar repetir un

servicio, cada solicitud emplea un número de secuencia único. El servidor mantiene la respuesta a cada solicitud hasta que él se asegure de que el cliente la recibió. Si el servidor llega a recibir una solicitud duplicada, éste la detecta mediante el número de secuencia, descarta el mensaje y reenvía la respuesta asociada.

3.2. Eliminación del reconocimiento de la respuesta

Tradicionalmente, el servidor requiere un reconocimiento a la respuesta para que este pueda liberar los recursos asociados a la respuesta. Nuestro enfoque es la utilización de un *timeout* que indique cuando se pueden liberar los recursos asociados a la respuesta. Este *timeout* es calculado en función del tiempo máximo de espera del cliente por la respuesta antes de que él concluya que el servidor se ha caído. De esta forma, se garantiza que no llegará una solicitud luego de que los recursos asociados a la respuesta han sido liberados. Así pues, las respuestas no son reconocidas.

3.3. Servicios de larga duración

Servicios de larga duración pueden causar expiración de *timeouts* asociados a la solicitud. El servidor resuelve este problema empleando un *timeout* que expira antes del *timeout* del cliente. Este *timeout* le indica al servidor que debe reconocer explícitamente la solicitud del cliente con el fin de evitar reenvios. Esto puede causar que el cliente permanezca bloqueado para siempre si el servidor se cae luego de enviar el reconocimiento. Para evitar esto, el cliente utiliza un protocolo de estetoscopio en el cual el cliente envía periódicamente al servidor solicitudes latido para cerciorarse que él no se ha caído.

3.4. Caídas del cliente o del servidor

Una caída del servidor puede causar que se pierdan los números de secuencia de los mensajes recibidos. Además, puede que no se completen algunos de los servicios. Si el servidor reencarna, el cliente puede reenviar una solicitud que será aceptada por el servidor, siendo en realidad una solicitud repetida. Para solucionar este problema, se usan épocas en el servidor. Cada vez que el servidor se “cae” y reencarna, la época es incrementada, de manera que el servidor deseche mensajes de épocas anteriores. Cuando el servidor recibe un mensaje con época anterior, este lo rechaza pero notifica el cambio de época al cliente.

Si el cliente envía una solicitud y se “cae” antes de recibir la respuesta, se tiene una solicitud “huérfana”. Si el cliente luego reencarna, se pueden tener repuestas antiguas que pueden causar inconsistencias en el cliente. Para resolver este problema, se usan épocas en el cliente. El cliente rechaza respuestas “huérfanas” basándose en su época.

4. Implantación

El sistema está compuesto por:

- Una biblioteca que utiliza cada proceso usuario del servicio.
- Un servidor de comunicaciones por sitio, que maneja el intercambio de mensajes remotos.

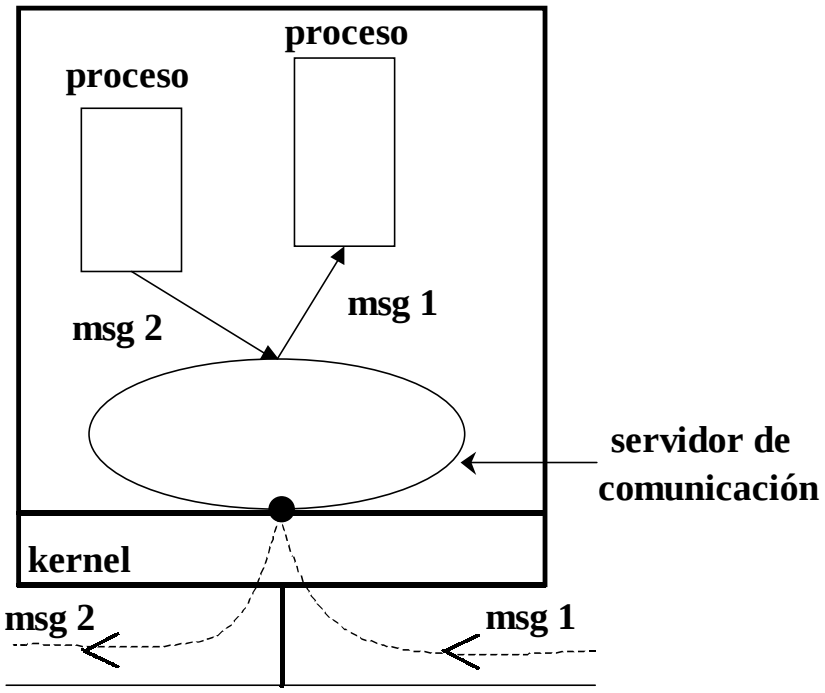

Figura 3.Sistema de Comunicación.

La comunicación entre los procesos usuario y el servidor de comunicación es implantada con colas de mensajes *System V*. El rol de la librería es enviar y recibir mensajes del servidor. En la Figura 3 se muestra como es la comunicación utilizando nuestro sistema.

A continuación se describe la arquitectura y los componentes del servidor.

4.1. Arquitectura

El servidor de comunicación posee un estilo arquitectónico muy sencillo basado en abstracción de datos y organización orientada por objetos, compuesta por cuatro módulos que exhiben un mínimo acoplamiento y que poseen la mayor cohesión posible. Adicionalmente, existen algunas estructuras de datos compartidas por los módulos, tales como tablas hash, donde se almacena información requerida por el servidor: números de secuencia, identificadores de mensajes, etc. La figura 4 ilustra la arquitectura del servidor.

Los módulos del servidor son los siguientes:

- **I/O Local**: Encargado de la comunicación local entre el servidor y los procesos.
- **Receive**: Este componente recibe los mensajes a través de la red. Este es el componente que "dirige" el protocolo, pues se encarga de manejar la mayoría de las situaciones de comunicación. Tales situaciones son: recepción de solicitudes, recepción de respuestas, inserción de tiempos de retransmisión, cancelación de retransmisiones, actualización de números de secuencia y rechazo de mensajes duplicados.
- **Timer**: Este componente es el encargado de manejar el tiempo. El es utilizado para activar las acciones relacionadas con los ***timeouts*** cuando estos expiren.

- **Send**: Este componente se encarga de enviar los mensajes a través de la red. Este es el componente más simple. Su única tarea es utilizar el servicio de UDP para enviar los mensajes al sitio que se le especifique.

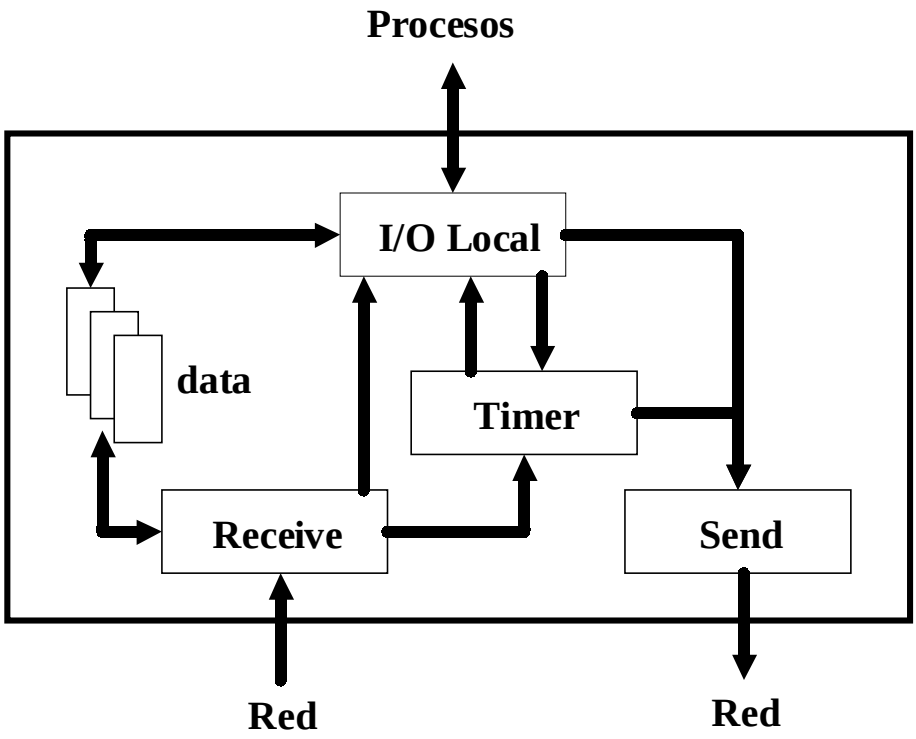

Figura 4. Arquitectura del Servidor.

4.2. Estructura de los modulos

El resto de esta sección describe los componentes del servidor del sistema. Cada componente posee un hilo y esta implantado por una clase C++. Cabe destacar que ninguno de los hilos del servidor consume ciclos CPU de espera ocupada en las acciones concernientes a sus actividades. En pocas palabras ninguno de los hilos hace "*polling*".

4.2.1. Módulo de entrada y salida (I/O Local)

Este componente provee la comunicación entre el servidor de comunicación y los procesos. El servidor tiene asociada una cola de mensajes *System V*, donde los procesos colocan los mensajes. A su vez, cada proceso también tiene asociada una cola mensajes *System V* para recibir los mensajes.

I/O Local, utiliza a *Send* para enviar mensajes a través de la red e inserta los eventos de tiempo en el *Timer*. Por ejemplo, cuando un proceso envía una solicitud, *I/O local* lo recibe e inserta el *timeout* asociado a la retransmisión del mensaje en el *Timer*, luego realiza algunas operaciones en las tablas de datos y envía el mensaje a *Send* para que éste lo envíe al destino.

El componente contiene un hilo en escucha permanente que procesa los mensajes que se colocan en la cola de mensajes del servidor. La interfaz de este módulo con el resto de los módulos está definida por las siguientes primitivas:

- `void sendLocalRequest(const RequestMsg*):` Con este método se envían las respuestas a los procesos.
- `void sendLocalReply(const ReplyMsg*):` Se utiliza para enviar una solicitud a un proceso que presta servicio.
- `void timeoutExpired(const TimeoutMsg*)`: Se utiliza para indicarle al proceso que se ha alcanzado el tiempo máximo de espera por la respuesta.

4.2.2. Módulo de recepción de mensajes remotos (Receive)

Básicamente, este componente es un hilo que esta "escuchando" permanentemente a través de la red. Cada mensaje que llega es procesado por este hilo según el tipo de mensaje.

Receive inserta y cancela los eventos de tiempo en el *Timer* cuando llegan mensajes a través de la red. Además, se comunica con el *I/O Local* para que éste entregue mensajes a los procesos. Por ejemplo, cuando llega una respuesta a una solicitud, *Receive* cancela el *timeout* asociado a la retransmisión de la solicitud y envía esta respuesta a *I/O local* para que este último la entregue al proceso destino. Este componente no presenta ninguna interfaz al resto de los módulos del sistema.

4.2.3. Módulo de manejo de eventos temporales (Timer)

Este componente contiene un hilo que realiza las acciones asociadas a los tiempos que expiran. El evento más inmediato es gestionado mediante una cola de prioridad implantada con un "heap" binario. La cola almacena eventos que son insertados por los demás componentes. Las prioridades están dadas por los tiempos absolutos de los eventos. Cuando expira el tiempo absoluto de la primera entrada en la cola, las acciones asociadas al evento se ejecutan.

Timer es activado cada vez que expira un *timeout*. Además, *Timer* envía mensajes a *Send* para que éste último retransmita mensajes. Adicionalmente, *Timer* notifica a *I/O Local* la expiración de un *timeout* cliente. De este modo, *I/O Local* le notificará al cliente la expiración del tiempo máximo de espera por la respuesta. La interfaz de este módulo es la siguiente:

- `void insertEvent(Event*):` Este método inserta un evento en la cola de prioridad. Cada evento tiene asociado funciones a ejecutar cuando se alcance el tiempo absoluto. Por ejemplo, cuando se envía una solicitud, *I/O Local* inserta un evento asociado a un tiempo absoluto t. La función de este evento es retransmitir el mensaje al tiempo t.
- `void cancelEvent(Event*):` Este método permite cancelar un evento que esta en la cola de prioridad. Es utilizado cuando las acciones asociadas al evento ya no se deben ejecutar. Por ejemplo, si llega una respuesta, el evento cuya acción es retransmitir la solicitud es cancelado.

4.2.4. Módulo de envío de mensajes remotos (Send)

Este componente posee una cola FIFO donde se colocan los mensajes a enviar a través de la red. Un hilo, permanentemente, toma los mensajes de la cola y los envía a través de la red. El componente *Send* no envía ningún tipo de información al resto de los componentes. La interfaz es la siguiente:

- `void sendMessage(MessageHeader* const)`: Esta método permite insertar mensajes en la cola FIFO para que sean enviados a través de la red.

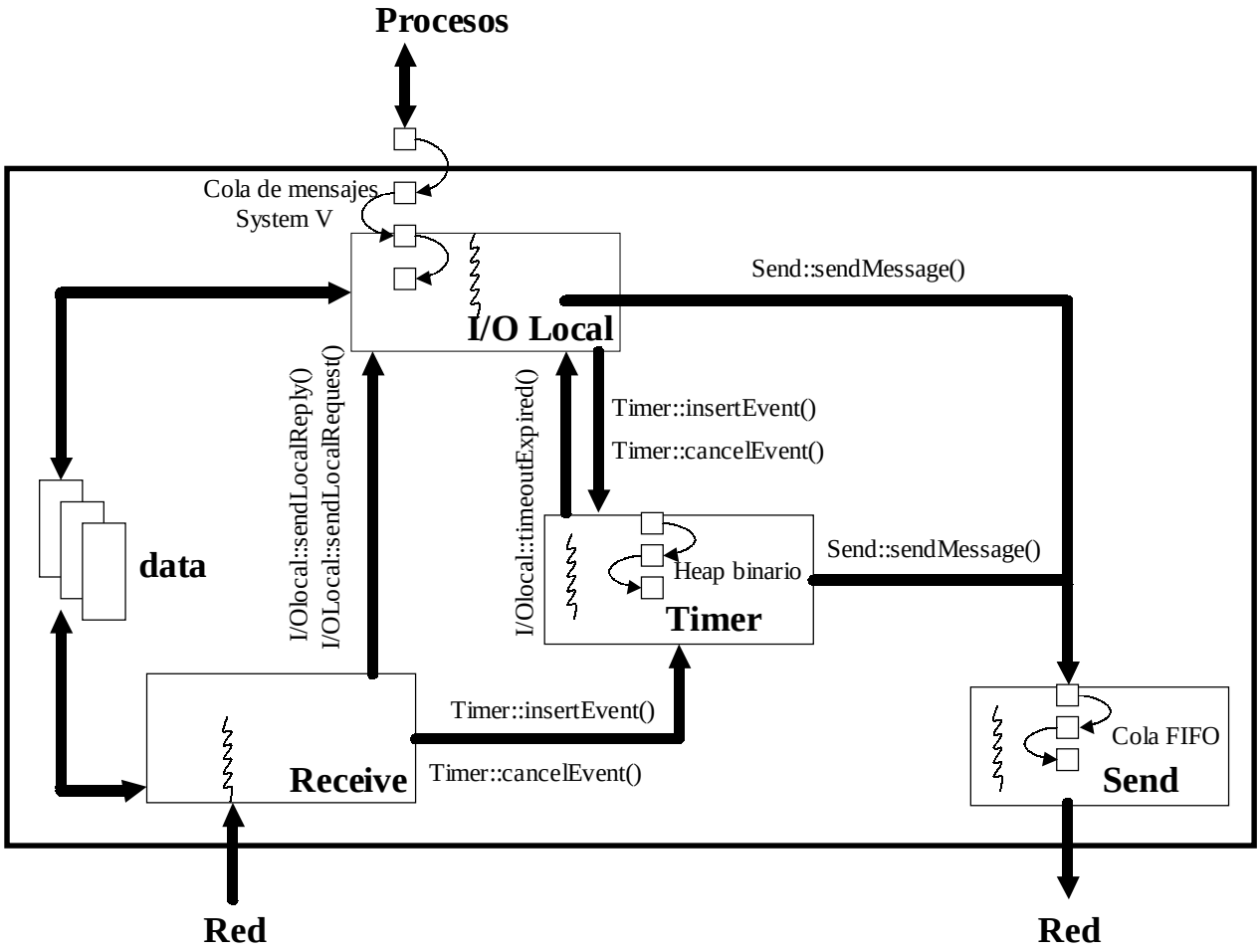

Figura 5. Componentes del Servidor.

5. Desempeño.

Como una primera evaluación, nosotros comparamos nuestro sistema con TCP. El experimento consistió en medir el tiempo total de envió de solicitud y recepción de respuesta. Los tamaños de los mensajes se variaron desde 1 hasta 3.5 Mb. Por cada tamaño, se calculó el promedio de 1000 intercambios solicitud-respuesta. Esto fue realizado sobre una red LAN y una red MAN.

5.1. Red Local.

En este experimento se colocaron a el cliente y al servidor en dos maquinas distintas sobre una red LAN aislada. Las maquinas utilizadas fueron un Pentium MMX de 200 Mhz con 32 MB de memoria RAM, tarjeta Ethernet de 10Mb y una Pentium de 166 Mhz con 32 Mb de memoria RAM, tarjeta Ethernet de 10 Mb.

En la Figura 6 se muestran los resultados. La primera grafica muestra la latencia hasta 64 bytes para el tamaño del mensaje y la segunda muestra la latencia de 64 bytes en adelante.

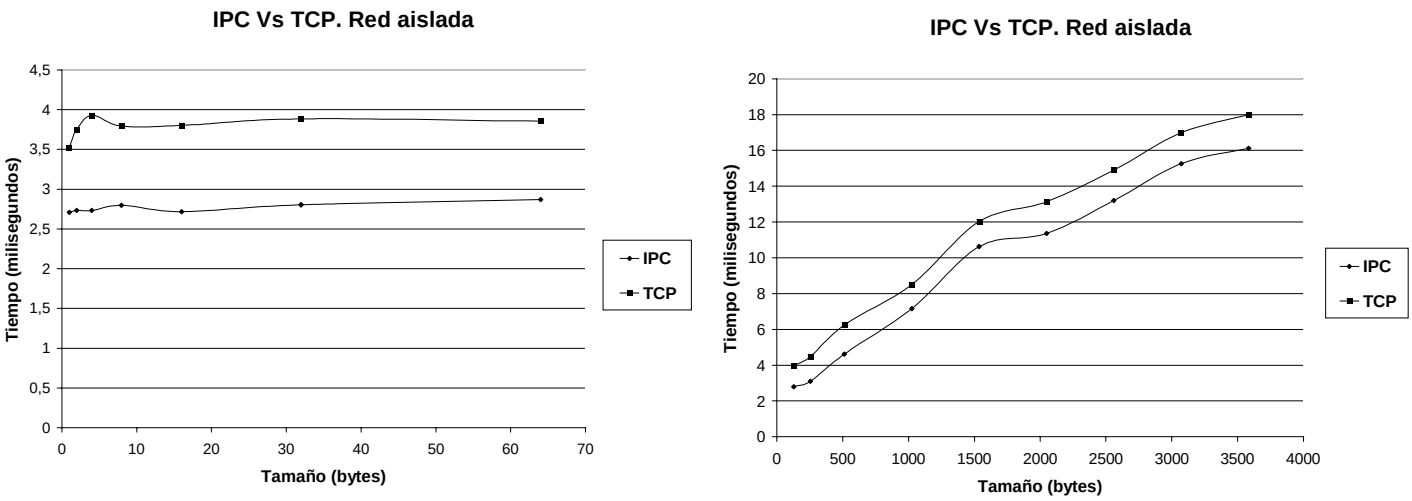

Figura 6. Desempeño para Red aislada.

Podemos observar que el protocolo IPC es más eficiente que TCP. El tiempo de solicitud–respuesta aumenta considerablemente cuando se utilizan mensajes de tamaño significativo. Puede observarse, además, que la diferencia de tiempo entre el IPC y TCP es constante. Esto se debe, probablemente, a que el tiempo empleado en establecer y cerrar una conexión es independiente de la cantidad de datos a transmitir.

5.2. Red Metropolitana

En este experimento se colocaron al cliente y al servidor en dos redes LAN distintas, interconectadas por la Red de área metropolitana de la Universidad de Los Andes. Cabe recalcar que las dos LAN estaban separadas por tres enrutadores. La maquinas utilizadas fueron un Pentium MMX de 200 Mhz con 32 MB de memoria RAM, tarjeta Ethernet de 10Mb y una 486 DX 33 Mhz con 32 Mb de memoria RAM, tarjeta Ethernet de 10 Mb.

En este experimento se obtuvo un comportamiento similar a 5.1 aunque, por supuesto, los tiempos aumentaron debido a las distancias entre las máquinas participantes.

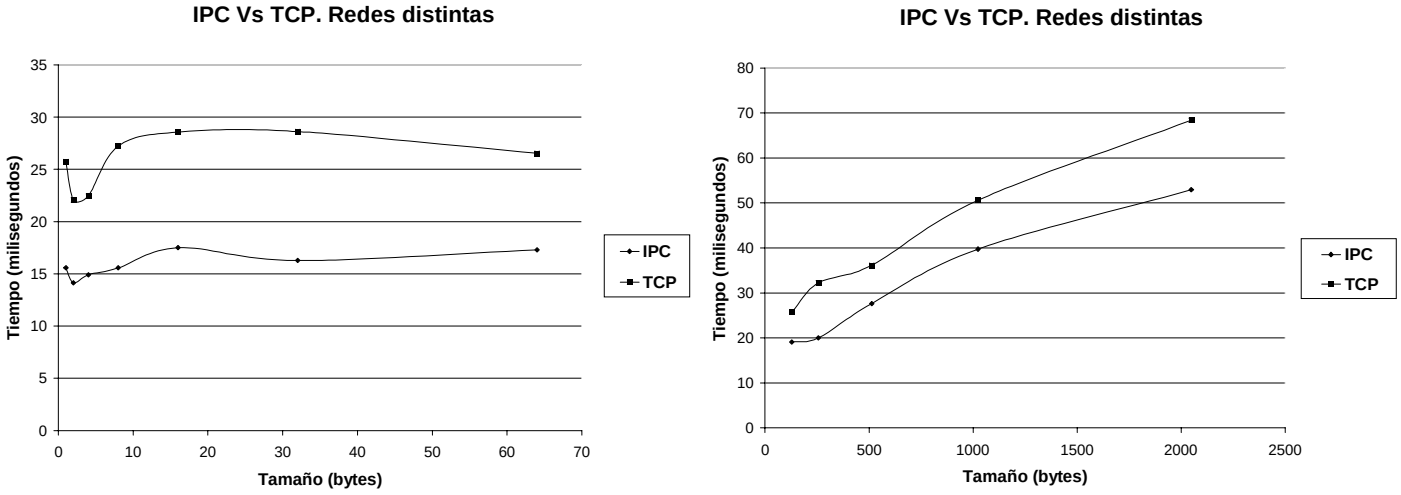

Figura 7. Desempeño, Redes distintas.

6. Futuro trabajo.

A partir de este sistema se tiene pensado desarrollar los trabajos futuros siguientes:

- Implantar MPI[5] y PVM[4] con nuestro sistema y efectuar medidas de rendimiento sobre aplicaciones reales que utilicen estos sistemas.
- Proveer soporte para mensajes mas grandes que un datagrama UDP.
- Extender el sistema para comunicación local. Nuestra interfaz permite ser utilizada (por especialización) para comunicación local. Esto permitiría utilizar un IPC local de alto desempeño basado en [1].
- Soporte para solicitudes anidadas con propagación del aborto. Es decir, cuando un servidor se convierte en un cliente para la misma solicitud que recibió, entonces es necesario propagar el aborto de una solicitud.
- Realizar pruebas de rendimiento con T/TCP, una extensión de TCP para transacciones. T/TCP es un protocolo que pretende eliminar el sobrecosto de comunicación de TCP y a su vez proveer confiabilidad en la comunicación.

7. Conclusión.

En Internet, el protocolo de transporte a usar, depende de la naturaleza de la comunicación. En aplicaciones donde existe un flujo continuo de datos, tal como FTP, es recomendable la utilización de TCP. Por otra parte, en aplicaciones donde existe un simple intercambio solicitud - respuesta se puede utilizar UDP, pero, la aplicación debe agregar la confiabilidad necesaria.

El legado de este trabajo puede resumirse en los siguientes comentarios:

- Se dispone de un sistema de comunicación genérico que permite desarrollar aplicaciones cliente – servidor. Este sistema provee todos los mecanismos necesarios para lidiar con los problemas de comunicación en red. Por lo tanto, este sistema puede ser utilizado para el desarrollo de aplicaciones en Internet. El sistema esta disponible vía FTP en ftp.cemisid.ing.ula.ve, directorio pub/aleph/ipc.
- El sistema exporta una interfaz sencilla de usar y que permite independencia de localización. Esta independencia se logra a través de la abstracción llamada puerto.
- El servidor de comunicaciones presenta una arquitectura muy sencilla, donde se muestra cada uno de los componentes necesarios para implementar el protocolo (§ 3). Cada uno de estos componentes exhibe una gran cohesión y un mínimo acoplamiento, esto permite sustituirlos o agregar cualquier optimización sin mayores inconvenientes.

8. Bibliografía.

[1] Bershad et al. Lightweight remote procedure call. Proceedings of the 12th ACM Symposium on Operating System Principles, 23(5), pp. 102-13, December 1989.

[2] Comer, Douglas. TCP/IP. Principios básicos, protocolos y arquitectura. Tercera Edición Prentice Hall. 1996. ISBN: 968-880-541-6.

[3] David Garlan and Mary Shaw. An Introduction to Software Architecture. Software Engineering Institute, Technical Report. Carnigie Mellon University. Pittsburg, jan 1994.

[4] Dongarra, J.J., Rempel, R. Hey, A.J., Walker D. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, feb. 1993.

[5] MPI Forum. MPI : A Message-Passing Interface. Technical Report, Department of Computer Science, Oregon Graduate Institute, Number CS/E 94-013, mar. 94.

[6] Nava, Carlos. Soporte de comunicación para sistemas de invocación a objetos. Tesis de grado, Universidad de Los Andes, dic. 1999.

[7] Nelson, B and A. Birrell. Implementing Remote Procedure Call. ACM Transactions on Computer Systems. Vol 1, No. 2, pp. 39-59: 1984.

[8] Panzieri, F. and Shrivastava, S.K. “Rajdoot: a remote procedure call mechanism with orphan detection and killing. IEEE Trans. On Software Engineering. vol 14, pp. 30-37, jan 1988.

[9] Stevens, Richard. TCP/IP Illustrated, Volume 1, The Protocols. Addison-Wesley.1994. ISBN: 0-201-63346-9.

[10] Stevens, Richard. UNIX Network Programming. Volume 1 Second Edition. Prentice Hall 1998. ISBN: 0-13-490012-X.

[11] Stevens, Richard. UNIX Network Programming. Volume 2 Second Edition. Prentice Hall 1999. ISBN: 0-13-081081-9.