FACULTAD DE INGENIERIA – INSTITUTO DE COMPUTACIÓN

REDES DE COMPUTADORAS

LABORATORIO Nº2

MATIAS PÉRES – NICOLAS VAZQUEZ - FEDERICO MUJICA – GERMAN RUIZ
OCTUBRE DEL 2012

1 ÍNDICE

2 INTRODUCCIÓN

En el presente trabajo se describe el proceso de resolución del laboratorio número dos de la asignatura Redes de Computadoras.

Al igual que en ocasiones anteriores, la realización del trabajo se ha enfocado de manera tal que sirva como guía para el lector, no solo como solución inmediata a las clausulas del obligatorio, sino que permita comprender el proceso en su completitud y la solución adoptada, que sea necesario, detallado y relevante en cuanto a las cláusulas establecidas.

Los objetivos son: cumplir una vez más con las cláusulas, ahondar en la utilización de los conceptos aprendidos, lograr un mayor manejo del entorno de desarrollo, ser lo más descriptivos posible de forma de lograr un desarrollo coherente, explicando las decisiones tomadas en cada sección del laboratorio y corregir o evitar errores surgidos durante el desarrollo del obligatorio anterior.

En el capítulo 3, se realiza un acercamiento al marco teórico de distintos conceptos cuyo significado es necesario desarrollar en la resolución del obligatorio, entre otras herramientas necesarias. Entre estos destaca, Modelo OSI, protocolo de conexión Three-Ways Handshake y la instancia específica del protocolo ARQ, Go-Back-N.

En el capítulo 4 se describe la resolución del obligatorio. Se realizan las observaciones necesarias en cuánto a detalles técnicos, siguiendo las bases establecidas en la teoría e introduciendo cambios y decisiones. Es importante señalar que en ningún momento se introducen cambios ajenos a las cláusulas establecidas por el obligatorio. El capítulo 5 responde a las preguntas planteadas en la hoja del obligatorio y finalmente en el capítulo 6 se realizan distintas pruebas con el fin de verificar el correcto funcionamiento de la aplicación.

El trabajo se encuentra respaldado por material bibliográfico confiable que ha sido seleccionado con atención, a efectos de que la información en la cual nos basemos para la realización del mismo sea certera, lo que brinda constantemente una seguridad en la lectura.

El proceso de elaboración del trabajo consistió en la división temática del mismo, asignando a cada uno de los integrantes del equipo una o varias secciones determinadas, teniendo en cuenta que el conocimiento sea general y llegando a una puesta en común antes de la división.

3 CONCEPTOS PREVIOS

3.1 MODELO OSI

El modelo OSI (Open System Interconnection) fue creado en el año 1984 por la Organización Internacional de Estándares (ISO) y es una base importante para las redes actuales. Antes de que la ISO impusiera este modelo, se dificultaba la comunicación entre redes de distintos fabricantes. Debido a esto la ISO quiso crear un modelo en el cual todos los fabricantes se adhirieran al modelo de red y así los dispositivos de red pudieran interactuar entre sí. Podemos establecer una analogía entre el modelo OSI y un lenguaje único el cual todas las redes deben hablar para poder comunicarse y entenderse.

Este proceso tan complejo de comunicación de dispositivos se pudo dividir en varias partes, de hecho en 7 capas en total, las cuales establece el modelo OSI.

Las capas 1, 2 y 3 son llamadas capas inferiores, la capa 4 es una capa intermedia y las capas 5, 6 y 7 son llamadas capas superiores, las que están "más cerca" del usuario.

Veamos una breve síntesis de cada una de estas capas:

Capa 7 – Capa de Aplicación: En esta capa las aplicaciones tienen la posibilidad de acceder a los servicios de las demás capas y monitorearlas mediante su dirección IP. También aquí se definen los protocolos utilizados por las aplicaciones para intercambiar datos.

Capa 6 – Capa de Presentación: Esta capa actúa como un traductor ya que debe garantizar que la información que envía la capa de aplicación de un sistema pueda ser leída y entendida por la capa de aplicación de otro sistema. Por lo tanto en esta capa se maneja la semántica y sintaxis de los datos transmitidos.

Capa 5 – Capa de Sesión: Es la capa que administra las sesiones entre dos maquinas que están comunicadas, procurando mantener la comunicación entre ellas mientras estén en red.

Capa 4 – Capa de Transporte: Esta capa es la encargada de efectuar el trasporte de datos de desde el host de origen al destino, utilizando uno de los 2 protocolos: TCP (confiable) o UDP (no confiable).

Capa 3 – Capa de Red: Es la capa encargada de verificar que los datos lleguen desde el host origen al destino, aun si estos no están conectados directamente.

Para esto cuentan con los routers, quienes trabajan en esta capa y se encargan de enrutar paquetes, determinando rutas o mejores caminos.

Capa 2 – Capa de Enlace: Proporciona la transmisión de datos a través del medio físico, recibiendo peticiones de la Capa de Red y utilizando servicios de la Capa Física, asegurándose que la información fluya sin errores entre las maquinas conectadas.

Capa 1 - Capa Física: Se encarga de definir la transmisión de la computadora hacia la red especificando el medio físico en el cual se realiza. Este medio puede ser guiado (por ejemplo fibra, cable coaxial) o no guiado (como por ejemplo las microondas o los infrarrojos). También esta capa establece el método de transmisión binaria (en unos y ceros) a través de los medios físicos.

3.2 CONCEPTOS TEORICOS PROTOCOLO TCP

3.2.1 Three-Ways Handshake

En esta sub-sección vamos a ver como se establece una conexión TCP, esto tiene una gran importancia, porque el establecimiento de una conexión TCP puede aumentar significativamente el retardo percibido (por ejemplo cuando se navega en la red). Muchos de los ataques de red más comunes, entre ellos el ataque por inundación de SYN, explota algunas vulnerabilidades de la gestión de la conexión TCP.

Vamos a entender cómo se establece una conexión usando la técnica Three-Ways Handshake, supongamos que un proceso en ejecución (Cliente) desea iniciar una conexión con otro proceso (Servidor), se ejecutan los siguientes pasos:

1) El cliente envía un segmento TCP especial al TCP del lado del servidor, Este segmento en especial no contiene datos, pero uno de los bits indicadores de la cabecera del segmento, el bit SYN se pone a 1. Por eso este segmento especial se referencia como segmento SYN. El cliente también ingresa de forma aleatoria un número de secuencia inicial (client_nsi) y lo coloca en el número de secuencia del segmento TCP. Este segmento se encapsula dentro de un data diagrama IP y se envía al servidor. El motivo por el cual se elige aleatoriamente el número de secuencia es para evitar ciertos ataques de seguridad.

2) Una vez que el data diagrama IP que contiene el segmento SYN TCP llega al host servidor, el servidor extrae dicho segmento SYN del datagrama, asignando buffers y variables TCP a la conexión y envía un segmento de conexión concedida al cliente TCP. Este segmento también especial no contiene datos, sino que contiene 3 fragmentos de información importantes en su cabecera. El primero el bit SYN se pone en 1, el campo de reconocimiento de cabecera (campo ACK) se le ingresa client_nsi + 1, y por último el servidor elige de forma aleatoria su propio número de secuencia inicial (server_nsi) y almacena este valor en el campo de secuencia de la cabecera IP. Este segmento TCP se conoce como SYNACK.

El cliente al recibir el segmento SYNACK, también asigna buffers y variables a la conexión. El cliente entonces envía también otro segmento; este último segmento confirma el segmento de conexión enviado por el servidor (el cliente hace esto almacenando el valor servidor_nsi+1 en el campo de reconocimiento de la cabecera del segmento TCP). El bit SYN se pone en 0, ya que la conexión ya está establecida.

3.3 GO-BACK-N ARQ

Go-Back-N ARQ es una instancia específica del protocolo ARQ (automatic repeat request), donde el proceso emisor puede transmitir varios paquetes (si están disponibles) sin tener que esperar a que sean reconocidos en el canal. A continuación se describe el protocolo GBN como se especifica en [2].

Definimos un número *base* como el número de secuencia del paquete no reconocido más antiguo y *signumsec* como el número de secuencia más pequeño no utilizado (es decir, el número de secuencia del paquete que se va a enviar). Se puede identificar entonces cuatro intervalos en el rango de números de secuencia. Los números de secuencia pertenecientes al intervalo [0, base-1] corresponden a paquetes que ya han sido enviados pero todavía no se han reconocido. El intervalo [base, signumsec-1] corresponde a paquetes que ya han sido enviados pero todavía no se han reconocido. Los números de secuencia del intervalo [signumsec, base+N-1] se pueden emplear para los paquetes que pueden ser enviados de forma inmediata, en caso de que lleguen datos procedentes de la capa superior. Y por último, los números de secuencia mayores o iguales que base+N no pueden ser utilizados hasta que un paquete

no reconocido que se encuentre actualmente en el canal sea reconocido (específicamente, el paquete cuyo número de secuencia sea igual a base).

El rango de los números de secuencia permitidos para los paquetes transmitidos pero todavía no reconocidos puede visualizarse como una ventana de tamaño N sobre el rango de los números de secuencia. Cuando el protocolo opera, esta ventana se desplaza hacia adelante sobre el espacio de los números de secuencia. Por esta razón, N suele denominase tamaño de ventana y el propio protocolo GBN se dice que es un protocolo de ventana deslizante.

En la práctica, el número de secuencia de un paquete se incluye en un campo de longitud fija de la cabecera del paquete. Si k es el número de bits contenido en el campo que especifica el número de secuencia del paquete, el rango de los números de secuencia será [0,2^k -1]. Con un rango finito de números de secuencia, todas las operaciones aritméticas que impliquen a los números de secuencia tendrán que efectuarse utilizando aritmética en módulo 2^k. El espacio de números de secuencia puede interpretarse como un anillo de tamaño 2^k, donde el número de secuencia 2^k -1 va seguido por el número se secuencia 0.

El emisor del protocolo GBN tiene que responder a tres tipos de sucesos:

- Invocación desde la capa superior: cuando se llama a rdt_enviar() desde la capa superior, lo primero que hace el emisor es ver si la ventana está llena; es decir, si hay N paquetes no reconocidos en circulación. Si la ventana no está llena, se crea y se envía un paquete y se actualizan las variables de la forma apropiada. Si la ventana está llena, el emisor simplemente devuelve los datos a la capa superior, indicando de forma implícita que la ventana está llena. Probablemente entonces la capa superior volverá a intentarlo más tarde. En una implementación real, muy posiblemente el emisor almacenaría en el buffer estos datos (pero no los enviaría de forma inmediata) o dispondría de un mecanismo de sincronización (por ejemplo, un semáforo o un indicador) que permitiría a la capa superior llamar a rdt_enviar() sólo cuando la ventana no estuviera llena.
- Recepción de un mensaje de reconocimiento ACK. En nuestro protocolo GBN, un reconocimiento de un paquete con un número de secuencia n implica un reconocimiento acumulativo, lo que indica que todos los paquetes con un número mayor o igual que n han sido correctamente recibidos por el receptor.

- Un suceso de fin de temporización. El nombre de este protocolo, “Retroceder N”, se deriva del comportamiento del emisor en presencia de paquetes perdidos o muy retardados. Como en los protocolos de parada y espera, se empleará un temporizador para recuperarse de la pérdida de datos o de reconocimiento de paquetes. Si se produce un fin de temporización, el emisor reenvía todos los paquetes que haya transmitido anteriormente y que todavía no hayan sido reconocidos. Si se recibe un paquete ACK pero existen más paquetes transmitidos adicionales no reconocidos, entonces se reinicia el temporizador. Si no hay paquetes no reconocidos en circulación, el temporizador se detiene.

Las acciones del receptor en el protocolo GBN también son simples. Si un paquete con un número de secuencia n se recibe correctamente y en orden (es decir, los últimos datos entregados a la capa superior preceden de un paquete con el número de secuencia n-1), el receptor envía un paquete ACK para el paquete n y entrega la parte de los datos del paquete a la capa superior. En todos los restantes casos, el receptor descarta el paquete y reenvía un mensaje ACK para el paquete recibido en orden más recientemente. Observe que dado que los paquetes se entregan a la capa superior de uno en uno, si el paquete k ha sido recibido y entregado, entonces todos los paquetes con un número de secuencia menor que k también han sido entregados. Por tanto, el uso de confirmaciones acumulativas es una opción natural del protocolo GBN.

En nuestro protocolo GBN, el receptor descarta los paquetes que no están en orden. Aunque puede parecer algo tonto y una pérdida de tiempo descartar un paquete recibido correctamente (pero desordenado), existe una justificación para hacerlo. Recuerde que el receptor debe entregar los datos en orden a la capa superior. Supongamos ahora que se espera el paquete n, pero debe llegar el paquete n+1. Puesto que los datos tienen que ser entregados en orden, el receptor podrá guardar en el buffer el paquete n+1 y luego entregar ese paquete a la capa superior después de haber recibido y entregado el paquete n. Sin embargo, si se pierde el paquete n, tanto él como el paquete n+1 serán retransmitidos como resultado de la regla de retransmisión del protocolo GBN en el lado de emisión. Por tanto, el receptor puede simplemente descartar el paquete n+1. La ventaja de este método es la simplicidad del almacenamiento en el buffer del receptor (el receptor no necesita almacenar en el buffer ninguno de los paquetes entregados desordenados. Por tanto, mientras el emisor tiene que mantener los límites inferior y superior de la ventana y la posición de signumsec dentro de esa ventana, el único fragmento de información que el receptor debe mantener es el número de secuencia del siguiente paquete en

orden. Este valor se almacena en la variable numsecesperado. Por supuesto, la desventaja de descartar un paquete correctamente recibido es que la siguiente retransmisión de dicho paquete puede alterarse y, por tanto, ser necesarias aún más retransmisiones.

4 RESOLUCIÓN

4.1 INTRODUCCIÓN

El laboratorio consiste en el desarrollo de un servicio de capa de transporte confiable y orientado a conexión el cual denominaremos PCT (Protocolo confiable de transporte), sobre el protocolo de capa de red Internet Protocol (IP). El término confiable nos indica que los datos enviados por el extremo origen, serán entregados a la aplicación destino sin pérdidas, y en el mismo orden en el que fueron enviados, independientemente de los problemas que puedan presentar la transferencia de datos por la red (pérdida o duplicación).

Características generales de la conexión *PCT*:

- Se debe establecer entre un extremo activo que inicia una conexión utilizando el método conectarPCT (cliente) y un extremo pasivo que espera conexiones utilizando el método aceptarPCT (servidor).
- Se enviarán datos en forma unidireccional desde el extremo que inició la conexión hacia el otro extremo. Para ello se proveen dos métodos escribirPCT() y leerPCT(), que implementan solicitud de envío y recepción respectivamente, entre las aplicaciones conectadas en los extremos.
- Se permite el cierre de conexiones, que podrá ser solicitado por cualquiera de los extremos (aunque no es posible ser solicitado directamente por la capa superior en caso del servidor), transmisión de bytes de un extremo al otro (tanto texto como de datos binarios), se respeta orden de entrega. Los datos se entregan a la capa de aplicación como un flujo continuo de bytes recibidos.
- Se implementan buffers de recepción y envío auxiliares, que son utilizados por las aplicaciones para escribir/leer los bytes enviados/recibidos.
- Se implementa un mecanismo de ARQ *Go Back N* descrito en el capítulo 3, tomándose de una constante predefinida (GBN_WINDOW) el tamaño de la ventana. En caso de saturación de los buffers de envío y/o recepción del *PCT*, se rechaza la solicitud de envío de la aplicación, o de recepción desde el extremo opuesto. Como se explicó se descartar los datos recibidos en forma duplicada.
- Solo puede existir una sesión *PCT* por aplicación. Las sesiones *PCT* se identifican con los siguientes 4 elementos, {*IP origen*, *puerto PCT origen*, *IP destino, puerto PCT destino*}. El *puerto PCT origen* debe tener un valor diferente al *puerto PCT destino*.

En el siguiente capítulo se describe la arquitectura de la aplicación.

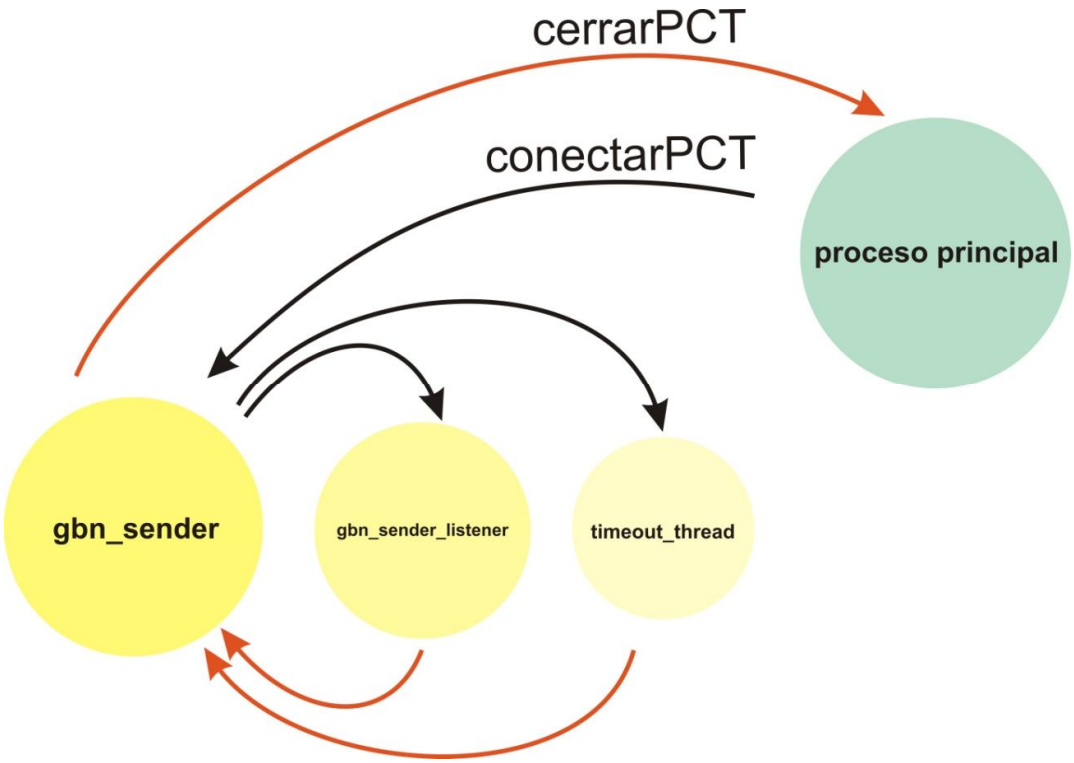
4.2 ESTRUCTURAS

Implementar una API en C/C++, que contenga las funciones definidas anteriormente. Dicha API se linkeditará con programas que utilicen las mencionadas funciones.

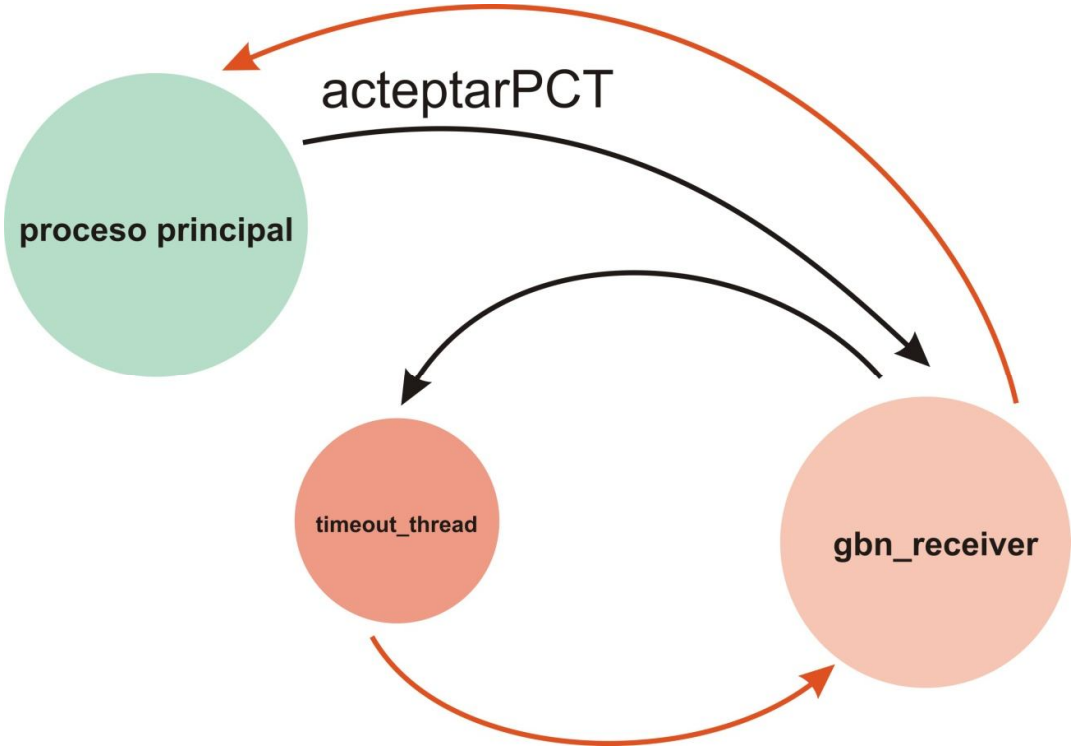
Como es posible entrever de la descripción anterior, se tendrá un gran módulo o componente que gestionará de forma independiente pero sincronizada los distintos pedidos solicitados por la capa superior.

Una vez comprendido los pasos necesarios para dar lugar a la creación de la aplicación, desarrollamos una arquitectura capaz de soportar dicho comportamiento pero que a su vez, permitiera la separación entre distintas funcionalidades del sistema, facilitando el trabajo en equipo y el desarrollo seguro hacia una aplicación sólida y eficiente. Otro de los aspectos tenidos en cuenta fue el desarrollo de un código comprensible y coherente, cuyo testeo no imponga dificultades ajenas a la intuición.

En primer lugar examinaremos dos diagramas que permiten comprender la creación y muerte de threads que se da lugar en la aplicación. Más adelante analizaremos la función de cada hilo de ejecución. Por el momento nos concentraremos en la estructura adoptada para su mantención.



Cuando se establece la conexión del lado del cliente, se crea un thread que ejecuta el procedimiento gbn_sender(). Este a su vez, crea dos threads más, gbn_sender_listener y timeout_thread. Al ejecutarse cerrarPCT, se despierta al thread gbn_sender() en caso de que este durmiendo y este espera a que mueran los dos threads que creó.

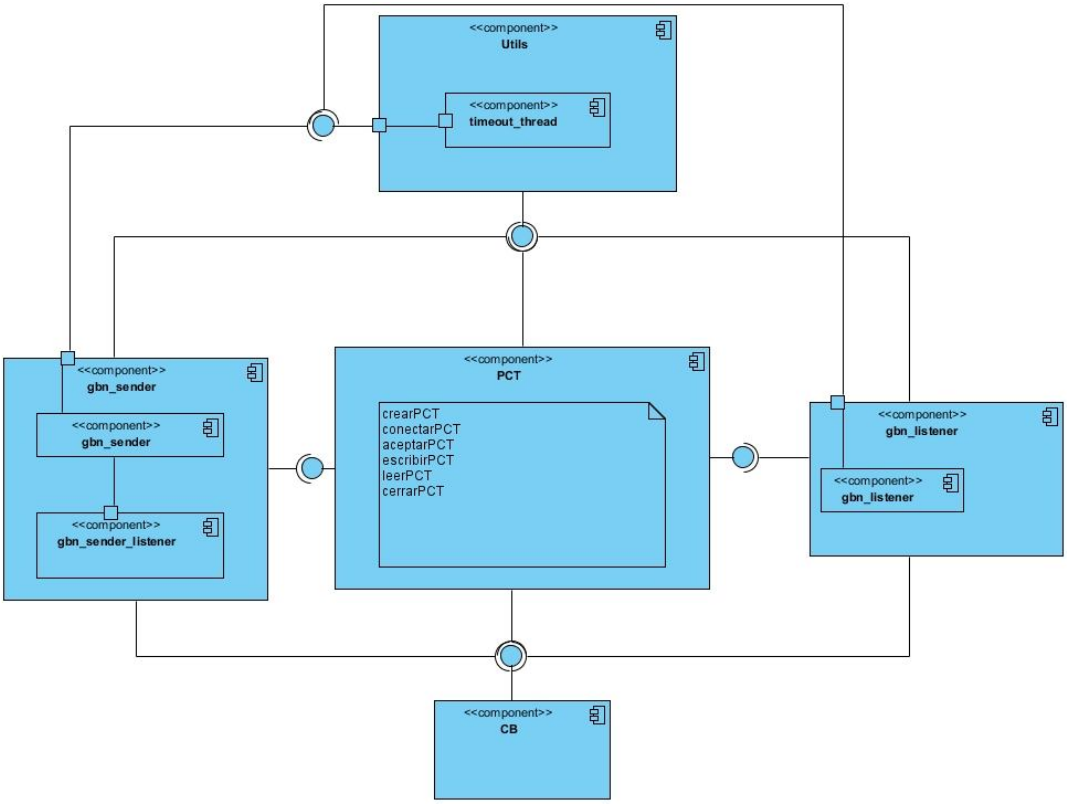


Cuando se establece la conexión del lado del servidor, se crea un thread que ejecuta el procedimiento gbn_receiver(). Este procedimiento a su vez, crea un thread que ejecuta timeout_thread().

Dado que se trata de una aplicación multihilo, se tienen varias instancias de ejecución accediendo a los mismos datos que eventualmente podrían interceptarse y ocasionar fallas de memoria, por lo que se necesita también del uso de semáforos

Para la presentación de la arquitectura se ha optado por un diagrama de componentes y conectores. Cada elemento del diagrama, como es usual en un diagrama de componentes, se manifiesta durante la ejecución del programa (como ser objetos o librerías), consume recursos y contribuye con el comportamiento del sistema.

En la siguiente imagen se muestra el diagrama realizado. El diagrama muestra instancias, no tipos. Su realización contribuyo al desarrollo del obligatorio, ayudando a visualizar el camino de la implementación, permitiendo tomar decisiones tempranas. Se debe tener en cuenta que se trata de un diagrama primitivo, cuya semántica puede no estar perfectamente definida. En particular, esta vista ayuda a responder la pregunta de cuáles son las entidades más importantes durante la ejecución de la aplicación y cómo interactúan.



En el diagrama anterior es posible ver los principales componentes de la aplicación. Se observa el componente PCT, con las funcionalidades provistas por la API. Los componentes gbn_sender y gbn_listener contienen los procedimientos de los threads principales creados al realizarse la conexión. Estos tres componetes hacen uso del componente CB (Circular buffer) que contiene las estructuras del buffer de recepción (buffer de datos) y buffer de salida (buffer de paquetes).

El componente utils contiene funciones comunes a todos los componentes, y el procedimiento timeout_thread, utilizado para mantener activa la conexión.

4.3 CREARPCT

Para poder comenzar a utilizar la API del PCT se invoca a esta función con la dirección ip local de la aplicación. Crea las estructuras de datos, e inicia los procedimientos necesarios para el control del *PCT*. Devuelve un valor mayor o igual a 0. En caso de error devuelve -1.

Se crea el raw socket y se guardan su descriptor y la ip pasada como variables globales. Luego se cambia el estado a NOCONECTADO y se inicializan variables y estructuras como el buffer y los semáforos para multiexcluir el acceso al mismo.

4.4 ESTABLECER CONEXIÓN

4.4.1 Utiles

Para la solución del problema se crearon 3 operaciones auxiliares que se utilizan en diferentes partes de la implementación.

1) *int make_pkt(unsigned char localPCTport, unsigned char peerPCTport, struct in_addr localIP,struct in_addr peerIP, char *payload, int payl_size, unsigned char flags_pct, unsigned char nro_SEC_pct,unsigned char nro_ACK_pct, char**packet);*
    - Esta funcion crea un paquete, dado el puerto local, el puerto al cual se envia el paquete, la ip local, la ip a la cual se envia el paquete, los datos que el paquete lleva, las flags, numero de secuencia y numero de ACK.

2) *void to_sockaddr(struct sockaddr_in *addr, struct in_addr ip);*
    - Esta funcion setea en el sockaddr_in la direccion ip**.**

3) *int esperarSYN_ACK(int socket_desc,unsigned char flag_chk,unsigned char localPCTport,struct sockaddr_in &src_addr, struct in_addr localIP ,struct pct_header *pkt_pct, struct iphdr * pkt_ip,int &recibiPKT);*
    - Esta función espera un paquete y luego realiza validación, la dirección de la cabecera ip debe coincidir con la ingresada en la variable LocalIp, el protocolo en la cabecera ip debe ser 0xFF, en el pct_header del paquete el puerto destino debe ser igual a el puerto ingresado en localPCTport y la flag debe coincidir con flag_chk**.**

4.4.2 ConectarPCT

int conectarPCT(unsigned char localPCTport,unsigned charpeerPCTport, struct in_addr peerIPaddr)

"Inicia en forma activa la conexión con otro equipo (peerIPaddr) a través del puerto PCT especificado (peerPCTport). Esta función bloquea la aplicación hasta establecer la conexión. En caso de realizar en forma satisfactoria el procedimiento de conexión devuelve 0, en caso contrario devuelve -1."

Se inicializan las variables y se genera el número de secuencia aleatorio.

El procedimiento comienza enviando un paquete SYN, creándolo con la operación make_pkt() para luego ser enviado a través de la función sendto().

```
char *packet;

//Creo el paquete SYN
int syn_size = make_pkt(localPCTport, peerPCTport, localIP, peerIPaddr, NULL,
0, SYN_FLAG, seq, 0x00, &packet);
Envio_PKT = sendto(socket_desc, packet, syn_size, 0,(struct sockaddr *)
&daddr, (socklen_t)sizeof (daddr));
```

Luego que se envía exitosamente el paquete, se debe saber si el socket está listo para recibir, para eso se utiliza la función select ();

Esta función cheque dado el socket_descriptor de un socket, si este está listo para escribir, leer o tiene alguna condición excepcional pendiente (en nuestro caso nos interesa si está listo para leer). También se le puede setear un timeout que en el caso que sea null, select () se bloquea indefinidamente hasta que el socket esté listo, sino espera un hasta que el timeout se agote.

Con la función *esperarSYN_ACK* se espera recibir el paquete SYNACK, si este es recibido correctamente seteamos los valores de secuencia y ack.

```
struct pct_header *pkt_pct = (struct pct_header*) malloc(sizeof (struct
pct_header));
struct iphdr *pkt_ip = (struct iphdr*) malloc(sizeof (struct iphdr));

rec = esperarSYN_ACK(socket_desc, (SYN_FLAG | ACK_FLAG), localPCTport,
src_addr, localIP, pkt_pct, pkt_ip, recibiOK);
```

Por último si el paquete SYNACK es correcto se genera y envía el segmento ACK y se da como establecida la conexión.

Esta estructura esa controlada mediante un bucle do – while que controla el time out correspondiente.

4.4.3 aceptarPCT

int aceptarPCT(unsigned char localPCTport)

"Queda esperando en forma pasiva la conexión de otro equipo a través del puerto PCT especificado (localPCTport). Esta función bloquea la aplicación hasta recibir una conexión. En caso de realizar en forma satisfactoria el procedimiento de conexión devuelve 0, en caso contrario devuelve -1."

Se inicializan las variable y enseguida se pasa a el estado “Esperar SYN”

```
//Les reservo memoria a las estructuras que van a ser seteadas en
esperarSYN_ACK
struct pct_header *pkt_pct = (struct pct_header*) malloc(sizeof (struct
pct_header));
struct iphdr *pkt_ip = (struct iphdr*) malloc(sizeof (struct iphdr));

rec = esperarSYN_ACK(socket_desc, SYN_FLAG, localPCTport, src_addr, localIP
,pkt_pct, pkt_ip, recibiPKT);
```

Una vez que el paquete es recibido con éxito y validado, se crea el paquete SYNACK y se envía a la dirección de fuente que vino en el paquete SYN.

Luego utilizando la función select() compruebo que el socket esté listo para leer.

Cuando esté listo para leer paso a el estado esperando ACK.

```
rec = esperarSYN_ACK(socket_desc, ACK_FLAG, localPCTport, src_addr, localIP,
pkt_pct, pkt_ip, recibiPKT);
```

Si el paquete recibido es validado, se establece la conexión.

Esta estructura esa controlada mediante un bucle do – while que controla el time out correspondiente.

4.5 GBN_SENDER

Una vez implementado las funcionalidades relacionadas con el establecimiento de la conexión se procedió a desarrollar los procedimientos que permiten a nuestra aplicación comportarse de acuerdo al protocolo Go-Back-N ARQ desarrollado en el capítulo 3. De esa descripción se puede extraer una clara idea del funcionamiento del protocolo y surge inmediatamente la necesidad de lograr una implementación que imite de forma exacta el comportamiento.

Como se estableció, se disponen de tres tipos de sucesos a los cuáles se debe responder. Recordemos:

- Invocación desde la capa superior: cuando se llama a rdt_enviar() desde la capa superior. Este procedimiento se corresponde con el procedimiento escribirPCT(const voi*buf, size_t len) del cabezal pct.h. Ahora bien, en este procedimiento recae una diferencia fundamental con el comportamiento descrito más arriba, dado que se decidió adoptar una implementación real y aceptar datos, aún cuando con estos no quepan en

la ventana, o mejor dicho, aún cuando hay N paquetes no reconocidos en circulación. Si la ventana no está llena, se crea un paquete con los datos, se inserta el paquete en el buffer y luego se envía a través de la red. Si la ventana está llena pero no así el buffer de paquetes, cuyas dimensiones son preferiblemente mucho más grandes que la ventana, se crea un paquete y es insertado en el buffer. Dicho paquete no es enviado a través de la red, dado que, como se indica en la descripción del protocolo, los números de secuencia mayores o iguales que base+N no pueden ser utilizados hasta que un paquete no reconocido que se encuentre actualmente en el canal sea reconocido. Si no hay lugar en el buffer de paquetes para 1 paquete más, entonces se rechazan los datos. Más adelante veremos cuando se envían los paquetes que fueron guardados en el buffer, pero no enviados. Ahora examinemos el pseudicódigo de escribirPCT():

```
int escribirPCT(const void *buf, size_t len)
{

    …

    P(buff_mutex); // Multiexclusión

    Si buffer de paquetes lleno, retornar 0.

    // ARMO PAQUETE CON LO QUE QUEPA DE DATOS
    packet_size = make_pkt(…);

    // Lo meto en el buffer
    CB::Inst()->push_packet(packet, packet_size);

    if(CB::Inst()->CBcount() <= GBN_WINDOW) // Si el paquete entro en la ventana
    {
        if(CB::Inst()->CBcount() == 1) // Si es el primer paquete de la ventana
        {
            // Inicio temporizador
        }

        V(buff_mutex);

        // ENVIO PAQUETE
        if (sendto(…) < 0)
           // Manejo de error
    }
    else
    {
        V(buff_mutex);
    }

    // Retorno los datos que se guardaron
    return data_size;
}
```

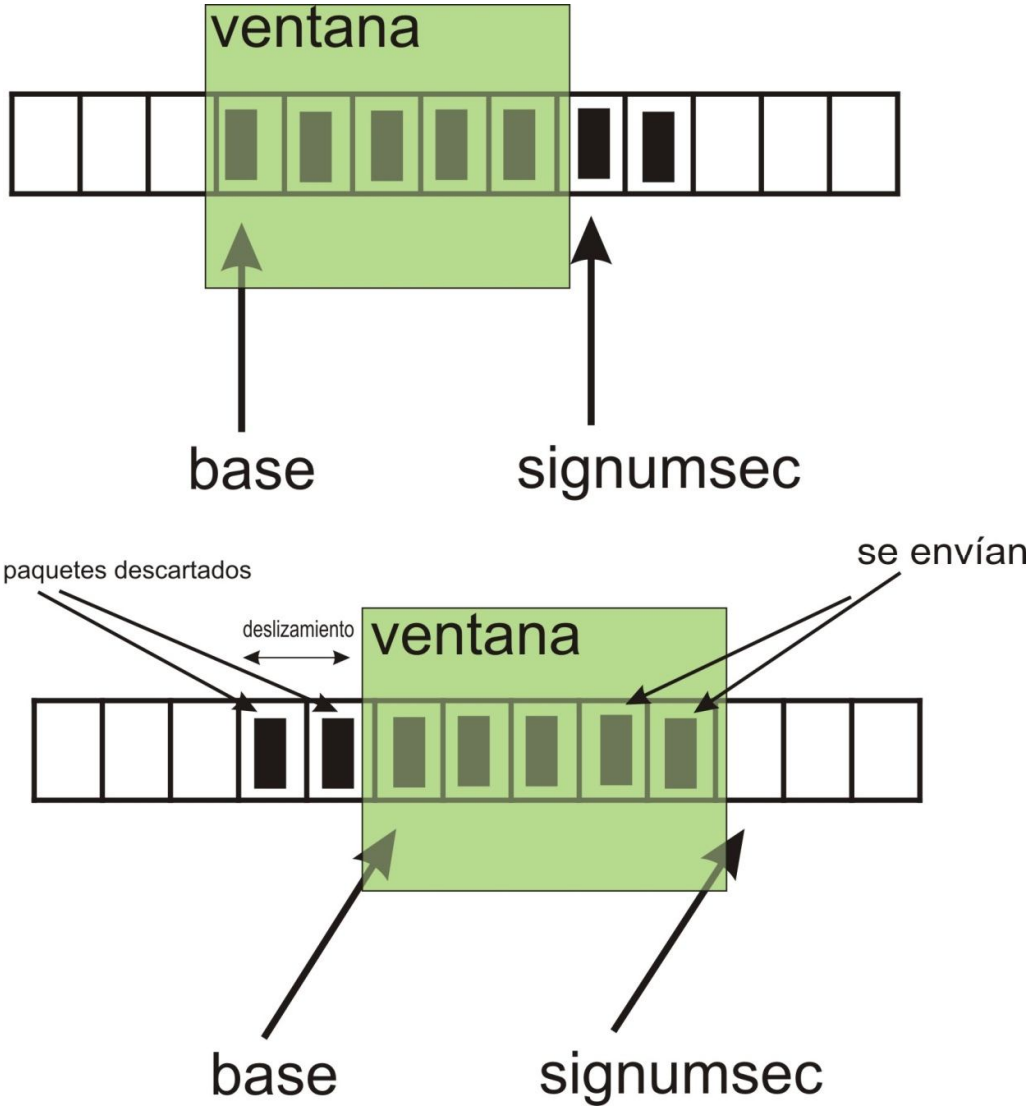
Segundo suceso al cuál se debe responder:

- Recepción de un mensaje de reconocimiento ACK. Como se mencionó, en el protocolo GBN, un reconocimiento de un paquete con un número de secuencia n implica un reconocimiento acumulativo, lo que indica que todos los paquetes con un número mayor o igual que n han sido correctamente recibidos por el receptor y por lo tanto se debe quitar del buffer, lo que se traduce en un corrimiento de la ventana. Observemos que los paquetes que se encuentran dentro de la ventana son paquetes que aún no han sido reconocidos, es por ello que es necesario correr la base de la ventana para que siga conteniendo paquetes no reconocidos, de lo contrario, estos serán enviados nuevamente, o por aún, no podrán ser descartados jamás. Este procedimiento se corresponde en nuestro código con el procedimiento gbn_sender_listener(void* params), procedimiento ejecutado por un hilo que es lanzado del lado del cliente, una vez que se establece la conexión. Veamos su pseudocódigo:

```
void* gbn_sender_listener(void* params)
{

    …

    while (!fin)
    {
        /************ SELECT *************/

        La función select me permite dormir un tiempo máximo de 2*TIMEOUT
mientras espero por un paquete ACK.

        /*********************************/

    …

        /* Se comprueba si se ha enviado algo */
        if (FD_ISSET(socket_desc, &descriptoresLectura)) // Paquete recibido
        {

    …

            packet_size = recvfrom(…);
            if (packet_size < 0)
            {
                Manejo de error.
            }
            else
            {
                    if(Chequeo que el paquete recibido provenga de la instancia
              de aplicación con la que establecí conexión.)
                    {

                            // Reinicio el tiempo  pues recibí un paquete del otro
                    extremo

    …
```

```
                        /***************** RECIBI ACK ********************/
                        if (pkt_pct->flags_pct == ACK_FLAG)
                        {
                            P(buff_mutex);

                           // Tengo que eliminar del buffer hasta el ack obtenido
                    y tengo que envíar aquellos paquetes que entren en la ventana

                                if (base == signumsec)
                                    detener_temporizador
                                else
                                    iniciar_temporizador

                               V(buff_mutex);
                        }
                        /***************** ************ ********************/
                    }

                }
            }
        }
        else if(time_out.tv_sec == 0 && time_out.tv_usec == 0)
        {
              //Pasaron 2*TIMEOUT sin recibir paquetes, se termina la conexion
              estado = Estado de finalización de la aplicación.
    …
              fin = true; // Termino este thread
        }

    } /********* WHILE PRINCIPAL **********/
    pthread_exit(NULL); // Fin del thread
}
```

Observaciones: para corroborar que se trata de un paquete correcto se verifican valores del paquete recibido y de parámetros correspondientes a la función *recvfrom()*. Mencionamos en el punto anterior que nuestra implementación del protocolo GBN, busca ser una implementación fiel a la realidad, y que por dicho motivo se dispone de un buffer suficientemente grande como para permitir el ingreso de paquetes aún cuando estos no caben en la ventana. Los paquetes que ingresan en este estado no son enviados automáticamente sino que deben esperar a que se desocupe la ventana para ser enviados. Es por ello, que una vez se recibe un paquete ACK confirmando uno o más paquetes (paquetes descartados), y la ventana es deslizada dando lugar a otros paquetes, se deben enviar aquellos paquetes que se encontraban en el buffer y que aún no habían sido enviados. Siempre y cuando estos quepan en el nuevo espacio formado por el deslizamiento de la ventana. Observar el siguiente diagrama para comprender mejor la situación:



Otro punto importante que se debe destacar es la corrección del movimiento de la ventana. En el siguiente punto analizaremos el procedimiento que cumple la función de fin de temporización. Como se busca llegar a un máximo de concurrencia, la implementación adoptada evita la multiexclusión del buffer al enviar toda la venta. Ahora bien, dicho comportamiento converge a la siguiente situación: es posible que al estarse enviando toda la venta, se reciba un paquete ACK confirmando uno o más paquetes de la ventana que se encuentra en proceso de envío. Es por ello que se decidió un mecanismo que permite corregir el actual envío de la ventana y enviar solo los paquetes que aún no se han confirmado. Esta decisión nos permite aumentar la rapidez de transferencia y no generar inconvenientes a la hora de ir a buscar paquetes confirmados, dado que estos, al estar confirmados, pueden haber sido sobrescritos.

Si no hay paquetes no reconocidos en circulación, el temporizador se detiene.

Comentamos el tercer y último suceso:

- Un suceso de fin de temporización. Como en los protocolos de parada y espera, se emplea un temporizador para recuperarse de la pérdida de datos o de reconocimiento de paquete al producirse un fin de temporización. Se reenvían todos los paquetes que hayan sido transmitidos anteriormente y que todavía no hayan sido reconocidos, estos son los paquetes que se encuentran entre la base de la ventana y la base más N (tamaño de la ventana). Aunque la ventana puede no estar llena completamente. El procedimiento correspondiente es gbn_sender(void* params) que es ejecutado por un hilo independiente lanzado cuando se establece la conexión del lado del cliente. Analizamos su pseudocódigo:

```
void* gbn_sender(void* params)
{

    …

    while (!fin)
    {


       /************ SELECT *************/

        La función select me permite dormir un tiempo de TIMEOUT.

        /*********************************/


        /*********** CERRANDO CONEXION *****************/
        if(estado == estado de cierre de conexión)
        {
            Analizado más adelante.

        } /*---------------- CERRANDO CONEXION --------------------*/
        else if( time_out.tv_sec == 0 &&  time_out.tv_usec == 0)
        {

    …

            P(buff_mutex);

            // Me ajusto al movimiento de la ventana
            i -= window_moved;
            window_moved = 0;
            if(i < 0) i = 0;

           // Leo el paquete del buffer
            packet_size = CB::Inst()->read (….);

            V(buff_mutex);
```

```
            if((packet_size > 0) && i < GBN_WINDOW)
            {
                  // ENVIO PAQUETE
                  if (sendto(…) < 0)
                        manejo de error
            }

            V(buff_mutex);
        }
    } /********* WHILE PRINCIPAL **********/

    pthread_exit(NULL);
}
```

Notar como no se multiexcluye el uso del buffer durante todo el envío de la ventana. Notar también el ajuste al movimiento del buffer dentro de la multiexclusión, lo que permite coordinación exacta entre los dos hilos. Algo importante que se debe mencionar en este punto es el cierre de la conexión. Cuando se ejecuta el cierre, este procedimiento es el encargado de despertar a los threads que se correspondan con el envío de información y que hayan nacido a causa de la conexión. Se espera a que los threads terminen. Además se envía todo lo que se encuentra en el buffer. Se envía un paquete FIN y se espera un FIN ACK. Si en un tiempo de TIMEOUT no llega, el thread termina de todas formas y la aplicación pasa a un estado de no conexión.

```
// Despierto a los threads
…

// Espero que los threads terminen
pthread_join(timer_thread, NULL);
pthread_join(thread_gbn_sender_listener, NULL);

// Envio todo lo del buffer
for(i = 0; i < CB::Inst()->CBcount(); i++)
{
     packet_size = CB::Inst()->read (…);
     if (sendto(….) < 0)
          manejo de error
}


// Armo paquete FIN
holder_size = make_pkt(…);

// ENVIO FIN
if (sendto(…) < 0)
     manejo de error

// ESPERO FIN ACK. SINO LLEGA SALGO POR TIMEOUT
while( time_out.tv_sec > 0 &&  time_out.tv_usec > 0 && !finack)
{
     if (select(….) < 0)
          manejo de error
```

```
    if (FD_ISSET(socket_desc, &descriptoresLectura)) // Paquete recibido
    {

    …
        /***************** RECIBI FINACK ********************/
        if (pkt_pct->flags_pct == ACK_FLAG | FIN_FLAG)
        {
            finack = true;
        }
        /*--------------------- RECIBI FINACK -------------------------*/
    }
}
```

4.6 GBN_RECEIVER

Al aceptar la conexión utilizando la operación *aceptarPCT* de la API implementada, se crea este hilo que lo llamaremos receptor. La función de este hilo es recibir los paquetes que envía el otro "extremo" (hilo emisor), extraer los datos y escribirlos en el buffer de datos para que luego la aplicación con la operación *leerPCT* pueda leerlos.

Lo primero que se hace es crear el hilo que controla que los extremos siguen conectados. También se usa una variable de *timeout* con la función de controlar que se reciba alguna paquete en 2*TIMEOUT segundos. Como se especifica en la letra, si no se recibe nada en este tiempo se da la conexión como perdida. En el siguiente pseudocódigo se detalla el comienzo del receptor y la estructura principal del hilo:

```
void* gbn_receiver(void*)
{
// Se crea el hilo que controla la conexion
CrearHiloConexion(..)
// Estructura para utilizar en el recvfrom, para recibir los paquetes
struct sockaddr_in src_addr;
int fromlen = sizeof(src_addr);
// Para quedarse con el select esperando para leer del socket
fd_set reader;
// Para manejar el timeout, se inicializa con 2*TIMEOUT, el tiempo que se necesita
para dar la // conexión como perdida
struct timeval timeout;
timeout.tv_sec = 2*TIMEOUT;
timeout.tv_usec = 0;
// Headers ip y pct, para decodificar los paquetes recibidos
struct iphdr* packet_ip;
struct pct_header* packet_pct;
```

```
//Otras variables
fin = false;
While (!fin)
{
        RecibirPaquetes(…) //Si no recibo nada en 2*TIMEOUT => conexión perdida
        If (recibi && chekPaquete())
                procesarPaquete()
        else if (!recibi)
        {
                fin = true
                estado = NOCONECTADO //Variable global compartida por los hilos
        }
}
//Despierto al hilo de enviar y termina por la variable estado
pthread_join(thread_timer_sender, NULL);
// Salgo del hilo receptor
pthread_exit(NULL);
}
```

Como se puede observar se crea el hilo, se declaran e inicializan estructuras y variables a utilizar en el ciclo principal del hilo. Al entrar al ciclo lo primero que se hace es inicializar las variables necesarias para la función *select* que se utiliza para esperar la recepción de paquetes. Se setean el descriptor del socket donde se reciben los datos y además un "pipe" que utilizamos para despertar del *select* a los hilos escribiendo en ese "pipe" de manera similar a como se escribe en una archivo. Como ya se explico antes si no se despierta del *select* antes de que pasen 2*TIMEOUT segundos entonces se asume la conexión como perdida cambiando el estado y terminando el hilo como se puede ver en el pseudocódigo que muestra la estructura principal del hilo.

Al despertar del *select* se verifica que se escribió en el socket donde recibe los paquetes. Este checkeo se realiza para saber que no se paso el tiempo para dar la conexión por perdida. A continuación se lee del socket con la operación *recvfrom*, con el descriptor del socket donde se recibe, el cual es global a los hilos. Luego se pasa a decodificar el paquete extrayendo los cabezales ip y pct para poder verificar que el paquete recibido lo envió el otro extremo de la conexión. Para esto se comparan las direcciones ip destino, local y el protocolo ip (0xFF), que en nuestro caso seteamos al crear los paquetes. Luego se verifican los puertos en el cabezal pct. En el siguiente pseudocódigo se puede observar lo explicado:

```
select(….)
if (FD_ISSET(socket_desc, &reader)) //Si se escribio en el socket
```

```
{
        If (recvfrom(socket_desc, packet, …))
        {
                ip_header = getIpHeader(packet)
                if (checkIpHeader(ip_header)) // Se checkean las direcciones ip y el
protocolo
                {
                        pct_header = getPctHeader(packet)
                        If (checkPctHeader(pct_header)
                                extraerData(paquet)

                }
        }
}
//Timeout
```

Cuando finalmente se verifica que el paquete es enviado efectivamente por el otro extremo conectado se pasa a extraer los datos del paquete para poder escribirlos en el buffer de la aplicación receptora.

El hilo receptor puede recibir dos tipos de paquetes, uno con datos y otro para avisar el cierre de conexión. Para identificar el de datos se setea el atributo flags_pct con el valor de la bandera de data (DATA_FLAG) que es una constante definida en la aplicación. Para el paquete que indica el cierre de conexión se setea con el valor de la bandera de fin (FIN_FLAG).

Si el paquete resulta ser de datos se comparan el número de secuencia del mismo con el esperado. Al pasar la comparación se extrae los datos del paquete y se intentan escribir en el buffer. Si no hay espacio en el buffer entonces es como si el paquete se hubiera perdido, es decir como no se guarda no se aumenta el número de secuencia esperado dejando el paquete como no recibido.

```
If (esData(packet))
{
        entro = false
        If (numeroSecuenciaOk(packet))
        {
                data = getData(packet)
                entro = push_data_buffer(data)
        }
        unsigned char secNumAck;
          if (entro)
          {
```

```
            ultNum = num;
            secNumAck = num;
            mandar = true;
          }
          else
          {
            secNumAck = ultNum;
          }
          if (mandar)
          {
              char ACK_Packet[MAX_PACKET_SIZE];
              int packet_size = make_pkt(localPort, peerPort, localIP, peerIP, NULL, 0,
                                    ACK_FLAG, 0, secNumAck, ACK_Packet);
            sendto(socket_desc, ACK_Packet, packet_size, 0, (struct
                                                sockaddr*)&src_addr,
(socklen_t)sizeof(src_addr))

          }
}
```

Como se puede observar solo si el paquete tiene el número de secuencia igual al esperado se guardan los datos al buffer. Luego se pasa a mandar el paquete de confirmación ACK al extremo emisor. Si los datos entraron entonces se manda en el paquete ACK el número de secuencia esperado cuando se recibió el paquete. Si los datos no entraron se manda con el numero de secuencia del último paquete confirmado. En el caso especial que no hallan paquetes confirmados y no entren los datos no se arma el paquete ACK. Esto se controla con la variable *mandar*. Para enviar el paquete de confirmación se utiliza la operación *sendto* con el descriptor de socket y la dirección del mismo.

```
if (esFin(packet))
{
        // Mandar FIN,ACK y terminar la conexion
          char FINACK_Packet[MAX_PACKET_SIZE];
          int packet_size = make_pkt(localPort, peerPort, localIP, peerIP, NULL, 0,
         FIN_FLAG | ACK_FLAG, 0, packet_pct->nro_SEC_pct, FINACK_Packet);
         sendto(socket_desc, FINACK_Packet, packet_size, 0, (struct sockaddr
*)&src_addr, (socklen_t)sizeof(src_addr))

          estado = NOCONECTADO;
          fin = true
}
```

Si el paquete resulta ser de fin de conexión, entonces se arma un paquete con las banderas FIN,ACK y se manda al otro extremo confirmándole el final de conexión. Luego se cambia de estado y se finaliza el ciclo terminando el hilo.

4.7 CERRARPCT

Para finalizar la conexión PCT se invoca a esta función. Cierra la conexión en forma ordenada y destruye las estructuras de datos, y los procedimientos necesarios para el protocolo *PCT*. Previo al cierre deben enviarse los datos que se encuentran en los buffers de *PCT*. En caso de error devuelve -1.

La conexión la puede cerrar solo el que haya iniciado la conexión, es decir el extremo que invoco *conectarPCT.* Si es invocado por el extremo que acepto la conexión, el que vendría a ser el servidor que recibe los datos del cliente se retorna -1.

```
if (estado == CONECTADO)
   {
      estado = VACIANDOBUFFER;
      // Despierto al thread gbn_sender del select
     despertar(gbn_sender)

      //Espero que termine de vaciar el buffer y termine
      pthread_join(sender_thread, NULL);
      perror("El thread gbn_sender termino/n");

      estado = NOCONECTADO;
      //Libero memoria y cierro conexion
}
```

Entonces cuando el estado es CONECTADO, es decir que la aplicación es cliente, lo primero que se hace es cambiar el estado a VACIANDOBUFFER y despertar al hilo que se ocupa de enviar los paquetes. Con este estado especial se le avisa que se quiere cerrar la conexión y que debe enviar todo lo que se encuentre en el buffer.

Luego esta operación se bloquea esperando que termine el hilo vaciando el buffer. Se hace de esta manera para que el emisor no pueda escribir mas en el buffer mientras se está vaciando el mismo. Luego se cambia el estado a NOCONECTADO dando por finalizada la conexión y se pasa a liberar la memoria cerrando el socket y destruyendo los semáforos.

5 RESPUESTAS 2 Y 3 DE LA LETRA

5.1 ANÁLISIS DEL PROTOCOLO PCT CON VARIACIÓN DE TAMAÑO DE VENTANA.

Sin Interferencia:

Cuando no existe interferencia el comportamiento del protocolo no se ve afectado por el tamaño de la ventana.

Con Interferencia:

Al tener la ventana en tamaño 1, tengo que esperar a que me devuelvan un ACK para poder enviar otro paquete, en caso que el ACK del correspondiente paquete se pierda, tengo que esperar un time out para reenviar el paquete por que asumo que no llego.

La ventaja de Go-Back-N es que es sencillo de implementar y que el receptor no necesita disponer de un buffer de almacenamiento, puesto que sólo acepta paquetes entregados en orden. Sin embargo, por cada error, se produce la retransmisión de N paquetes, por lo cual cuando se produce un error se vuelve ineficiente. Mientras mayor sea el tamaño de la venta mayo va a ser la ineficiencia a la hora de recuperase de un error, ya que se debe realizar reenvíos de paquetes.

Por este motivo a medida que se aumenta el tamaño de la ventana, con la red configurada con interferencia aumenta el tiempo promedio de envío

5.2 FLUJO DE DATOS

Cuando una aplicación envía datos a TCP, lo hace en secuencias de bytes de 8-bits. Le corresponde entonces al envío de TCP para segmentar o delinear el flujo de bytes para transmitir datos en pedazos manejables para el receptor 1. Es esta falta de límites discográficas ", que le dan el nombre de" flujo de bytes de prestación de servicios.

El protocolo implementado, no soporta el “byte stream service”, la función escribirPCT() lo que hace es tomar la datos que le ingresan y crea un paquete con la totalidad de estos o de forma parcial en caso que los datos sea mayor al máximo tamaño permitido para enviar por paquete, también en el caso que no haya lugar en la ventana se envía mensaje de error.

Para poder soportar stream de bytes se debería implementar un buffer de datos, a medida que se le ingresen datos para enviar al protocolo PCT, se almacenan en dicho buffer, a la hora de crear un nuevo paquete para enviar se toma una "porción" de datos de dicho buffer y se ingresa este paquete a la ventana.

6 PRUEBAS REALIZADAS

En este capítulo mostraremos algunos ejemplos de pruebas que hemos realizado, para mostrar el comportamiento de nuestro programa en diferentes casos.

En primer mostraremos pruebas sin ejecutar el script netEmulator y luego ejecutándolo, para ver que se soporte la perdida de paquetes y duplicación de los mismos debidamente.

Las pruebas las realizamos corriendo dos terminales con derechos de administrador, mediante el comando sudo. En una terminar tendremos al emisor de datos y en la otra terminal al receptor de los mismos. Establecemos por defecto en este texto que la terminal 1 será la encargada de emitir datos y la terminal 2 quien los recibe.

En todas las pruebas utilizamos un buffer de tamaño 256.

- **enviaFile y recibeFile:**

Prueba 1)

VARIABLE	VALOR
TIMEOUT	30
GBN_WINDOW	25
MAX_PCT_DATA_SIZE	256
netEmulator.sh	down

Terminal 1: sudo ./enviaFile 3 127.0.0.3 1 127.0.0.1 logo_globedia.gif
Terminal 2: sudo ./recibeFile 1 127.0.0.1 recibido.gif

El archivo logo_globedia.gif pesa 5855 bytes, por lo tanto se requirió la transmisión de 23 paquetes de datos (22 de 256 bytes y el restante de 223 bytes). En este caso al no haber perdida de paquetes, cada vez que se envía un paquete de la terminal 1 es recibido en la terminal 2 y tiene el número de secuencia esperado, como vemos en este fragmento de consola:

Recibiendo DATA con secnum: 15 Esperado: 15
Recibido: 256 bytes
Enviando ACK packet con num: 15
Recibiendo DATA con secnum: 16 Esperado: 16
Recibido: 256 bytes
Enviando ACK packet con num: 16
Recibiendo DATA con secnum: 17 Esperado: 17
Recibido: 256 bytes
Enviando ACK packet con num: 17

La transmisión es realizada exitosamente.

Prueba 2)

VARIABLE	VALOR
TIMEOUT	30
GBN_WINDOW	25
MAX_PCT_DATA_SIZE	10
netEmulator.sh	down

Terminal 1: sudo ./enviaFile 3 127.0.0.3 1 127.0.0.1 logo_globedia.gif
Terminal 2: sudo ./recibeFile 1 127.0.0.1 recibido.gif

En este caso se requerirán 586 paquetes y como nuestro buffer es de tamaño 256 es de esperar que al trasmitir 256 paquetes se vuelva a reutilizar el buffer desde 0. Veamos que sucede en un momento crítico de la prueba que es cuando el buffer se llena:

Terminal 2:

Recibiendo DATA con secnum: 254 Esperado: 254
Recibido: 10 bytes
Enviando ACK packet con num: 254
Recibiendo DATA con secnum: 255 Esperado: 255
Recibido: 10 bytes
Enviando ACK packet con num: 255

```
Recibiendo DATA con secnum: 0 Esperado: 0
Recibido: 10 bytes
Enviando ACK packet con num: 0
Recibiendo DATA con secnum: 1 Esperado: 1
Recibido: 10 bytes
Enviando ACK packet con num: 1
Recibiendo DATA con secnum: 2 Esperado: 2
Recibido: 10 bytes
Enviando ACK packet con num: 2
```

Como vemos se realiza satisfactoriamente la circularidad del buffer, cuando este se llena se continua agregando nuevamente al principio. En esta prueba se "dio la vuelta" al buffer 2 veces y se logro transmitir satisfactoriamente el archivo.

La transmisión es realizada exitosamente.

Luego volvemos a realizar las pruebas 1 netEmulator.

Prueba 3)

VARIABLE	VALOR
TIMEOUT	30
GBN_WINDOW	25
MAX_PCT_DATA_SIZE	256
netEmulator.sh	up

Terminal 1: sudo ./enviaFile 3 127.0.0.3 1 127.0.0.1 logo_globedia.gif
Terminal 2: sudo ./recibeFile 1 127.0.0.1 recibido.gif

Al realizar nuevamente la prueba 1 nos encontramos con pérdida de paquetes y reordenamiento de paquetes, debido a la ejecución de netEmulator.

Terminal 2:

```
Recibiendo DATA con secnum: 18 Esperado: 18
Recibido: 256 bytes
Enviando ACK packet con num: 18
Recibiendo DATA con secnum: 21 Esperado: 19
Enviando ACK packet con num: 18
Recibiendo DATA con secnum: 19 Esperado: 19
Recibido: 256 bytes
Enviando ACK packet con num: 19
Recibiendo DATA con secnum: 20 Esperado: 20
Recibido: 256 bytes
Enviando ACK packet con num: 20
```

Como vemos en este fragmento, el receptor sigue enviando el ACK correspondiente al último paquete que esperaba recibir y recibió (el 18). Al llegarle el paquete 21 continua enviándole al emisor el ACK 18 y cuando recibe el paquete esperado envía el ACK correspondiente.

- **emisor y receptor:**

Prueba 4)

VARIABLE	VALOR
TIMEOUT	30
GBN_WINDOW	25
MAX_PCT_DATA_SIZE	256
netEmulator.sh	down

Terminal 1: sudo ./transmisor 3 127.0.0.3 1 127.0.0.1
Terminal 2: sudo ./receptor 1 127.0.0.1

En esta prueba no tenemos activado el script netEmulator, por lo tanto no hay perdida de paquetes en ningún caso, dejamos un ejemplo:

Terminal 1:

Escriba lineas y finalice con <enter>.<enter>:
hola
Prendo Ventana
Enviando (5 caracteres): >hola
<
como
Prendo Ventana
Enviando (5 caracteres): >como
<
estas
Prendo Ventana
Enviando (6 caracteres): >estas
<
?
Prendo Ventana
Enviando (2 caracteres): >?
<

Prendo Ventana

```
Enviando (1 caracteres): >
<
.
Prendo Ventana
Enviando (2 caracteres): >.
<
Transmisión finalizada
```

Terminal 2:

```
Recibiendo DATA con secnum: 0 Esperado: 0
Recibido: 5 bytes
Enviando ACK packet con num: 0
Recibidos (5 caracteres): >hola
<
Recibiendo DATA con secnum: 1 Esperado: 1
Recibido: 5 bytes
Enviando ACK packet con num: 1
Recibidos (5 caracteres): >como
<
Recibiendo DATA con secnum: 2 Esperado: 2
Recibido: 6 bytes
Enviando ACK packet con num: 2
Recibidos (6 caracteres): >estas
<
Recibiendo DATA con secnum: 3 Esperado: 3
Recibido: 2 bytes
Enviando ACK packet con num: 3
Recibidos (2 caracteres): >?
<
Recibiendo DATA con secnum: 4 Esperado: 4
Recibido: 1 bytes
Enviando ACK packet con num: 4
Recibiendo DATA con secnum: 5 Esperado: 5
Recibido: 2 bytes
Enviando ACK packet con num: 5
Recibidos (3 caracteres): >
.
<
```

Prueba 5)

VARIABLE	VALOR
TIMEOUT	30
GBN_WINDOW	25
MAX_PCT_DATA_SIZE	256
netEmulator.sh	up

Terminal 1: sudo ./transmisor 3 127.0.0.3 1 127.0.0.1
Terminal 2: sudo ./receptor 1 127.0.0.1

Volvemos a realizar la prueba 4, esta vez con el netEmulator activado, y obtenemos los siguientes resultados:

Terminal 1:

```
Escriba lineas y finalice con <enter>.<enter>:
hola
Prendo Ventana
Enviando (5 caracteres): >hola
<
como
Prendo Ventana
Enviando (5 caracteres): >como
<
estas
Prendo Ventana
Enviando (6 caracteres): >estas
<
?
Prendo Ventana
Enviando (2 caracteres): >?
<

Prendo Ventana
Enviando (1 caracteres): >
<
.
```

```
Prendo Ventana
Enviando (2 caracteres): >.
<
Transmisión finalizada
```

Terminal 2:

```
Recibiendo DATA con secnum: 0 Esperado: 0
Recibido: 5 bytes
Enviando ACK packet con num: 0
Recibidos (5 caracteres): >hola
<
Recibiendo DATA con secnum: 0 Esperado: 1
Enviando ACK packet con num: 0
Recibiendo DATA con secnum: 1 Esperado: 1
Recibido: 5 bytes
Enviando ACK packet con num: 1
Recibidos (5 caracteres): >como
<
Recibiendo DATA con secnum: 2 Esperado: 2
Recibido: 6 bytes
Enviando ACK packet con num: 2
Recibidos (6 caracteres): >estas
<
Recibiendo DATA con secnum: 2 Esperado: 3
Enviando ACK packet con num: 2
Recibiendo DATA con secnum: 3 Esperado: 3
Recibido: 2 bytes
Enviando ACK packet con num: 3
Recibidos (2 caracteres): >?
<
Recibiendo DATA con secnum: 4 Esperado: 4
Recibido: 1 bytes
Enviando ACK packet con num: 4
Recibiendo DATA con secnum: 5 Esperado: 5
Recibido: 2 bytes
Enviando ACK packet con num: 5
Recibidos (3 caracteres): >
.
<
```

7 CONCLUSIÓN

Hemos finalizado el informe sobre la realización del obligatorio número dos. Es importante mencionar los objetivos cumplidos.

El proceso hacia la formalización, ha tenido como propósito, según lo mencionado en ocasiones anteriores, ser coherentes en el desarrollo del informe, a diferencia de las ocasionales contradicciones que pueden encontrarse en piezas de información varias. Se pretendió fijar la atención en aspectos importantes y esenciales para la comprensión del desarrollo.

Podemos concluir que, al igual que en el obligatorio anterior, cumplimos nuestros objetivos cubriendo la totalidad de las cláusulas, implementando una aplicación con un fuerte uso de los nuevos conceptos aprendidos en el curso, logrando un mejor manejo del entorno de desarrollo y describiendo en la mejor forma posible la implementación logrando un desarrollo coherente y explicando las decisiones tomadas en cada sección del laboratorio.

Durante la preparación del trabajo los cambios y variantes, tanto de detalles como de perspectiva se hicieron numerosos y de múltiples estratos, sin embargo, en busca de una lectura confiable y coherente, se basaron las notas, observaciones y conclusiones en los textos bibliográficos.

Antes de la resolución del obligatorio, se intentó introducir al lector al aspecto teórico, y definir en forma precisa el vocabulario utilizado en etapas posteriores. Durante la explicación de la implementación se expusieron imágenes a modo de esclarecer la situación y agilizar la lectura. Se introdujo la arquitectura de la aplicación y se comentaron sus componentes principales.

En todo momento se realizaron pruebas que verificaran el correcto comportamiento de la aplicación.

Debemos mencionar que las decisiones tomadas a lo largo de la implementación surgieron como respuesta a los resultados obtenidos hasta el momento.

8 BIBLIOGRAFÍA

1. **Stalling, William.** *Comunicaciones y Redes de Computadoras.* Sexta edición.

2. **Kurose, James F. and Ross, Keith W.** *Computer Networking, A Top Down Approach.* Quinta Edición. 2010.

3. [En línea] Octubre de 2012. http://www.cs.umd.edu/~shankar/417-F01/Slides/chapter3a/sld029.htm.

4. [En línea] Octubre de 2012. http://courses.cs.vt.edu/~cs5516/spring03/slides/reliable_tx_1.pdf.

5. [En línea] Octubre de 2012. http://wps.aw.com/aw_kurose_network_4/63/16303/4173752.cw/index.ht.

6. [En línea] Octubre de 2012. http://en.wikipedia.org/wiki/Go-Back-N_ARQ.

7. [En línea] Octubre de 2012. http://www.chuidiang.com/clinux/sockets/socketselect.php.

8. [En línea] Octubre de 2012. http://www.chuidiang.com/clinux/senhales/senhales.php.

9. [En línea] Octubre de 2012. http://www.linuxquestions.org/questions/programming-9/c-linux-timers-how-223855/.

10. [En línea] Octubre de 2012. http://www.makelinux.net/ldd3/chp-6-sect-3.

11. [En línea] Octubre de 2012. http://pubs.opengroup.org/onlinepubs/009695399/functions/recvfrom.html.

12. [En línea] Octubre de 2012. http://pubs.opengroup.org/onlinepubs/009695399/functions/sendto.html.

13. [En línea] Octubre de 2012. http://www.mkssoftware.com/docs/man3/select.3.asp.

14. [En línea] Octubre de 2012. http://www.reloco.com.ar/linux/prog/pipes.html.

15. [En línea] Octubre de 2012. http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_join.html.