

Informe Obligatorio 2

Redes de Computadoras 2012

INCO – Facultad de Ingeniería – UdelaR
Octubre 2012

Grupo 59

Álvaro Acuña – CI: 3826062-8

Gabriel Centurión – CI: 2793486-8

Germán Mamberto – CI: 3187102-8

Fernando Mangino – CI: 3621009-1

Tabla de contenido

Objetivo	4
Descripción del Problema	4
Descripción del protocolo PCT	4
Formato de TPDU's utilizados por PCT	5
Descripción de la Solución	5
Descripción de la Implementación	7
Formato de paquetes enviados	8
Descripción de Raw Sockets	9
Establecimiento de la conexión	10
Descripción de Go Back N.....	14
Paquetes Keep Alive	15
Bibliotecas	15
Pseudocódigo.....	16
int crearPCT(struct in_addr localIPAddr).....	16
int aceptarPCT(unsigned char localPCTport)	16
int conectarPCT(unsigned char localPCTport,unsigned char peerPCTport, struct in_addr peerIPAddr).....	17
leerPCT:	18
escribirPCT:	18
cerrarPCT:.....	18
Preguntas	20
Pregunta1:.....	20
Pregunta2:.....	22
Manual de ejecución.....	23
Pruebas	24
Dificultades.....	32
Posibles Mejoras	32
Conclusiones	32

Objetivo

Implementar una API en C/C++, que contenga los métodos necesarios para la utilización de un protocolo de capa transporte confiable que denominaremos PCT.

Se debe resolver los siguientes elementos:

- Establecimiento de conexión del PCT, generando las estructuras sobre IP requeridas para la conexión real.
- Permitir el intercambio de datos en forma confiable, sin pérdida ni duplicación de datos manteniendo el orden de los mismos.
- Desconexión del PCT

Descripción del Problema

Se desea implementar un servicio de capa transporte confiable y orientado a conexión llamado PCT (Protocolo Confiable de Transporte), sobre el protocolo de capa red Internet Protocol (IP). El término confiable 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 pueda presentar la transferencia de datos por la red (pérdida o duplicación).

Descripción del protocolo PCT

El protocolo PCT debe realizarse sobre el protocolo de capa de red Internet Protocol (IP), el cual provee un servicio de datagramas no confiable, y será implementado por medio de raw sockets.

Solo existirá una sesión PCT por aplicación, las cuales se identificarán con [IP origen, puerto PCT origen, IP destino, puerto PCT destino].

En cuanto al establecimiento de la conexión:

- Se usará un mecanismo three-way handshake, donde un extremo activo inicia una conexión (conectarPCT) mientras el otro extremo pasivo espera conexiones (aceptarPCT).
- También se dará soporte para la verificación de que la conexión sigue activa mediante el envío de segmentos PCT Keep Alive.

En cuanto al envío de información:

- Se ofrecerán dos métodos (escribirPCT y leerPCT) que implementarán la solicitud de envío y recepción respectivamente.
- El envío será en forma unidireccional desde el extremo que inició la conexión hacia el otro extremo.
- El intercambio de información entre los dos extremos se realizará a través de TPDU's (Transport Protocol Data Unit) con formato PCT. Estos permitirán la transmisión de bytes de un extremo al otro (tanto texto como de datos binarios).
- Se implementarán buffers de recepción y envío auxiliares, que serán utilizados por las aplicaciones para escribir/leer los bytes enviados/recibidos.
- Para cumplir con la confiabilidad del servicio de transferencia de datos, PCT contará con una política Go Back N de Automatic Repeat reQuest (ARQ), garantizando así que los datos enviados por el cliente serán entregados a la aplicación destino sin pérdidas y en el mismo orden independientemente de los problemas que ocurran en la red como pérdida o duplicación.

En cuanto al cierre de la conexión:

- Se podrá cerrar la conexión desde cualquier extremo de la misma.
- Para lograr el cierre se enviará un segmento FIN, y se debe devolver FIN+ACK.
- En caso de pérdida del segmento FIN se saldrá por TIMEOUT.

Formato de TPDUs utilizados por PCT

El intercambio de información entre los dos extremos se debe realizar a través de TPDUs (Transport Protocol Data Unit) con formato PCT. El cabezal de los TPDUs es el presentado a continuación:

```
struct pct_header {  
    unsigned char srcPort;           // Puerto de origen  
    unsigned char destPort;         // Puerto Destino  
    unsigned char flags_pct;        // Flags de control  
    unsigned char nro_SEC_pct;      // Numero de secuencia de PCT  
    unsigned char nro_ACK_pct;      // Numero de ACK de PCT  
};
```

Descripción de la Solución

La propuesta de solución al servicio de capa transporte confiable será el desarrollo de una API que provea de los métodos necesarios para la comunicación de las aplicaciones.

- **int crearPCT(struct in_addr localIPaddr)** Crea las estructuras de datos, e inicia los procedimientos necesarios para el control del PCT.
- **int aceptarPCT(unsigned char localPCTport)** Queda esperando en forma pasiva la conexión de otro equipo a través del puerto PCT especificado (localPCTport).
- **int conectarPCT(unsigned char localPCTport, unsigned char peerPCTport, struct in_addr peerIPaddr)** Inicia en forma activa la conexión con otro equipo (peerIPaddr) a través del puerto PCT especificado (peerPCTport).
- **int escribirPCT(const void *buf, size_t len)** Solicita el envío al otro extremo de la conexión, de los datos pasados en buf de largo len. Esta función, dependiendo del estado y capacidad de los buffers disponibles, aceptará para envío hasta len bytes pasados en la variable buf, los que se almacenarán hasta tener la certeza de su recepción por el extremo opuesto.
- **int leerPCT(void *buf, size_t len)** Entrega como máximo len bytes recibidos por la conexión. En caso de no contar con datos devolverá 0. Los datos se devuelven a través del parámetro buf.
- **int cerrarPCT()** Cierra la conexión en forma ordenada y destruye las estructuras de datos, y los procedimientos necesarios para el protocolo PCT.

Es nuestra responsabilidad realizar la conexión de las aplicaciones, cuya implementación se basara en el mecanismo three-way handshake utilizado por TCP.

Como parte de la API a implementar se desarrollarán dos funciones (escribirPCT y leerPCT) que serían similares a las primitivas read y write utilizados en la tarea anterior. Pero en este caso es nuestra responsabilidad implementar y controlar su funcionamiento. Para cumplir con los objetivos y realizar un control de errores en la transmisión de datos se implementará el protocolo Go Back N, que nos permitirá en el caso de pérdida de paquetes el reenvío de estos, y el control de la recepción.

Descripción de la Implementación

El funcionamiento básico de nuestra solución es el siguiente:

La biblioteca desarrollada sirve de insumo para la comunicación entre dos aplicaciones, una de ellas con el rol de emisor y la otra de receptor.

Vale destacar que cada uno de los roles poseerá un buffer, de lectura en el caso del receptor, y uno de escritura en el caso del emisor.

El funcionamiento básico de cada rol seria:

Para el Emisor:

- Realiza la operación crearPCT de forma de inicializar las estructuras necesarias.
- Es el que realiza la operación conectarPCT, en la cual se realiza el intercambio necesario para establecer la conexión con el receptor, y además lanza en dos hilos independientes de ejecución las rutinas leeBufferyEnviaDatos y recepcionPaquetedeControl.
- La rutina leeBufferyEnviaDatos se encarga de ir leyendo los datos contenidos en el buffer de escritura e ir enviándolos al receptor en forma de paquetes de datos.
- La rutina recepcionPaquetedeControl se encarga de ir recibiendo los paquetes de control enviados por el receptor, como ser mensajes de ACK, FIN, FIN-ACK y KAL (KeepAlive). De forma de tomar las acciones pertinentes en cada caso.
- Realiza la operación escribirPCT mediante la cual ingresa al buffer de escritura los datos que desea enviar al receptor, los cuales serán leídos y enviados por la rutina leeBufferyEnviaDatos.
- Realiza la operación cerrarPCT la cual se encarga de realizar el intercambio necesario con el receptor de forma de cerrar la conexión, así como la destrucción de las estructuras inicializadas en el crearPCT

Para el Receptor:

- Realiza la operación crearPCT de forma de inicializar las estructuras necesarias.
- Es el que realiza la operación aceptarPCT, en la cual se realiza el intercambio necesario para establecer la conexión con el emisor, y además lanza en un hilo de ejecución independiente la rutina leeDatosYEnviaBuffer.
- La rutina leeDatosYEnviaBuffer se encarga de recibir los paquetes que son enviados por el emisor, e ir almacenándolos en el buffer de lectura siempre y cuando estos sean paquetes de datos y se trate de los que estábamos esperando. En caso de tratarse de paquetes de control (ACK, FIN, FIN-ACK y KAL) tomara las acciones pertinentes dependiendo del caso.
- Realiza la operación leerPCT mediante la cual extrae del buffer de lectura los datos enviados desde el emisor, los cuales serán leídos por la rutina leeDatosYEnviaBuffer.
- Realiza la operación cerrarPCT la cual se encarga de realizar el intercambio necesario con el receptor de forma de cerrar la conexión, así como la destrucción de las estructuras inicializadas en el crearPCT.

El cierre de la conexión, como se observó anteriormente, es realizado por la función cerrarPCT. Esta dependiendo del rol que la ejecute tendrá pequeñas variaciones.

Para el caso del Emisor, primero que nada se checkea si la conexión está establecida, en caso de que se encuentre desconectado, solamente realizara la destrucción de las estructuras para las cuales se poseía memoria reservada retornando -1. En caso de que aún se encuentre conectado, lo primero que hará es tener la precaución de aguardar que se hayan enviado todos los datos que se tenía en el buffer de escritura, así como la confirmación de la llegada de los mismos al receptor (ACKs). Luego de asegurarse que se haya enviado todo a lo que se comprometió enviara un paquete de control al receptor con la flag FIN, de forma de avisar al receptor que desea cerrar la conexión. Paso

siguiente se queda a la espera de la llegada del FIN-ACK. En caso de no llegar el mismo en un tiempo `timeoutFIN`, se enviara otro paquete FIN, repitiéndose este envío-espera una cantidad configurable de veces. En caso de que la llegada del FIN-ACK no se dé, se realiza la destrucción de las estructuras y se termina la ejecución retornando -1. En caso de que el FIN-ACK llegue antes de cumplidos todos los intentos, se destruyen las estructuras y se retorna 0.

Para el caso del Receptor, de forma análoga al Emisor lo primero que se realiza es el chequeo de la conexión, en caso de que se encuentre desconectado, solamente realizara la destrucción de las estructuras para las cuales se poseía memoria reservada, retornando -1. En caso de que aún se encuentre conectado, a diferencia del emisor, no realiza ninguna verificación sobre los datos del buffer. Solamente comenzara con el envío-espera del paquete FIN una cantidad configurable de veces. En caso de que la llegada del FIN-ACK no se dé, se realiza la destrucción de las estructuras y se termina la ejecución retornando -1. En caso de que el FIN-ACK llegue antes de cumplidos todos los intentos se destruyen las estructuras y se retorna 0.

Formato de paquetes enviados

Para el intercambio de información construimos datagramas IP con los datos necesarios en los campos de control y datos para cumplir con el formato PCT. Estos datagramas tendrán un header IP de 20 bytes y en su payload contendrán un header PCT de 5 bytes como el mencionado anteriormente (`pct_header`).

El formato de estos datagramas es el siguiente:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
Version				IHL				Type of Service								Total Length																0
Identification																Flags				Fragment Offset												1
Time to Live								Protocol								Header Checksum																2
Source Address																																3
Destination Address																																4
srcPort								destPort								flags_pct								nro_SEC_pct								5
nro ACK pct								data																								6

Dónde:

- Versión = 4, indicando que trabajaremos con IPv4.
- IHL = 5, indicando que el cabezal IP serán 5 palabras de 32 bits (20 bytes).
- Total Length = `sizeof (datagrama)`, ya que la cantidad de datos a enviar son de tamaño variable (entre 0 y `MAX_PCT_DATA_SIZE`).
- Time to Live = 64, valor por defecto
- Protocol = `IPPROTO_RAW`, ya que esta constante vale 0XFF (255) que es el numero elegido para identificar nuestro protocolo PCT.
- Source Address contendrá la dirección IP origen del datagrama.
- Destination Address contendrá la dirección IP destino del datagrama.
- ToS, Identification, Flags y Fragment Offset serán cargados con cero ya que no los utilizaremos.
- Header Checksum también será cero ya que para este obligatorio no se considera que los paquetes puedan llegar corruptos, por lo que si un paquete llega al destino, se supone que llega idénticamente a como partió.
- `srcPort` contendrá el puerto origen del datagrama.
- `destPort` contendrá el puerto destino del datagrama.

- `flags_pct` son flags de control del segmento PCT enviado e indica el tipo del segmento. Los tipos utilizados serán:
 - DATA_FLAG: representa un segmento PCT que contiene datos.
 - SYN_FLAG: representa un segmento PCT de control utilizado en el establecimiento de la conexión.
 - ACK_FLAG: representa un segmento PCT de control que indica que el segmento contiene información de reconocimiento.
 - FIN_FLAG: representa un segmento PCT de control utilizado en el fin de la conexión.
 - FLAG VACIA: representa un segmento PCT de control utilizado para verificar que la conexión continua activa (segmento keep alive).
- `nro_SEC_pct` indica el número de secuencia de PCT
- `nro_ACK_pct` indica el número de ACK de PCT
- `data` contendrán los datos a enviar que tendrán un tamaño entre 0 y `MAX_PCT_DATA_SIZE`.

Descripción de Raw Sockets

Como primera observación notamos que en la tarea anterior utilizamos un protocolo orientado a conexión (TCP) donde ambas aplicaciones debían necesariamente conectarse entre ellas mediante un socket y recién luego eran capaces de transmitir datos utilizando primitivas que proveía el sistema operativo como **read** y **write**, con lo cual se garantizaba la llegada correcta de todos los datos de una aplicación a la otra.

En cambio, en esta oportunidad muchas de estas cuestiones que ya resolvían dichas funciones las deberemos solucionar nosotros mismos ya que ahora utilizaremos funciones como **sendTo(...)** y **recvfrom(...)** sobre raw sockets, las cuales al no ser orientadas a conexión nos obligan a armar y verificar que los paquetes se envíen y reciban en orden y correctamente.

Como indica su nombre los “raw socket” son un nivel más abstracto que los sockets que utilizamos en la tarea anterior (`SOCK_STREAM`) que es un caso específico de `RAW SOCKET`.

Este tipo de sockets no está asociado con ningún protocolo de transporte en particular y nos brinda un mayor control ya que permite el acceso directo a protocolos de capas más bajas tales como IP, lo que nos permitirá crear los paquetes que necesitemos con sus headers específicos para así lograr nuestra implementación del PCT.

Como contra se pierde fiabilidad y será nuestra responsabilidad brindar la seguridad de que la información se transmita correctamente y sin pérdidas.

Algo a tener en cuenta es que los programas que usen “raw sockets” deberán ser ejecutados con permisos de root.

Una de las primeras cosas a tener en cuenta sobre los raw sockets es lo que comenta la letra de la tarea:

“Para la implementación, se utilizarán sockets del tipo Raw Sockets de la capa de red [2], que permiten el envío y recepción de datagramas, sin utilizar los mecanismos usuales aplicados por el sistema operativo. De ésta forma se tendrá acceso a todos los datagramas intercambiados en la interfaz de loopback (lo), siendo responsabilidad de la API diseñada la selección de los datagramas que tengan como destino la aplicación conectada por ella”

Por lo que al trabajar con raw socket en la interfaz de loopback, se puede dar el caso de leer el mismo paquete que estamos enviando. Por esta razón debemos controlar que solo consideremos los paquetes que vienen a nuestra dirección y puerto.

En cuanto a la creación de los sockets, utilizaremos la función `socket(AF_INET, SOCK_RAW, IPPROTO_RAW)`.

Con `AF_INET` indicamos que es un socket de red, con `SOCK_RAW` decimos que será un raw socket y en el lugar del protocolo se colocará la constante `IPPROTO_RAW`, cuyo valor es 255, para indicar que no nos asociaremos a ningún protocolo en particular, sino que recibiremos todos los paquetes independientemente del valor del protocolo.

Al trabajar sobre la interfaz de loopback, también se tendrá en cuenta que los puertos origen y destino del PCT deben ser diferentes .

Establecimiento de la conexión

Para establecer la conexión entre las dos aplicaciones se utilizará un procedimiento llamado negociación en tres pasos (three-way handshake).

Para realizar la conexión una aplicación (receptor) abre un socket y se queda escuchando. A esto se le llama apertura pasiva y determina el lado servidor de la conexión. En nuestra implementación la función `acceptarPCT(...)` será quien represente esta acción. Esta función bloquea la aplicación hasta recibir una conexión.

Por otro lado, la otra aplicación (emisor) hace una apertura activa abriendo un socket y enviando un paquete SYN (primer paso). En nuestra implementación la función `conectarPCT(...)` será quien represente esta acción. Esta función bloquea la aplicación hasta establecer la conexión.

Luego, el lado del servidor lo recibe y responde con un paquete SYN/ACK (segundo paso).

Finalmente, el cliente recibe este paquete y responde con un ACK, completando el proceso y estableciendo la conexión (tercer paso).

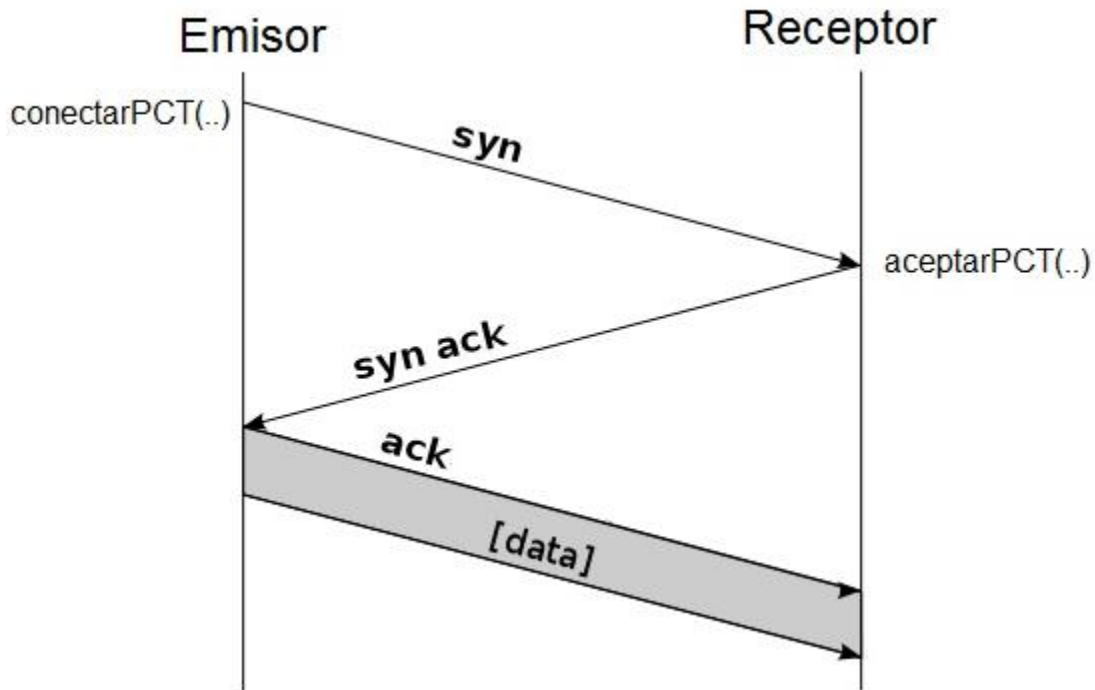
Utilizamos un numero de reintentos para el caso en que se pierdan los paquetes ACK o SYN/ACK , esto fue para soportar ruido en la red. Nuestra solución reenvia paquetes SYN si no le han llegado AYN/ACK, y reenvia SYN/ACK si no le llega el ACK.

Hay que tener en cuenta que en muchas aplicaciones web si no se logra la conexión al primer intento se devuelve error y se deja al usuario que intente nuevamente. Por lo que pensamos que estos reintentos también podrían hacer en la capa de aplicación.

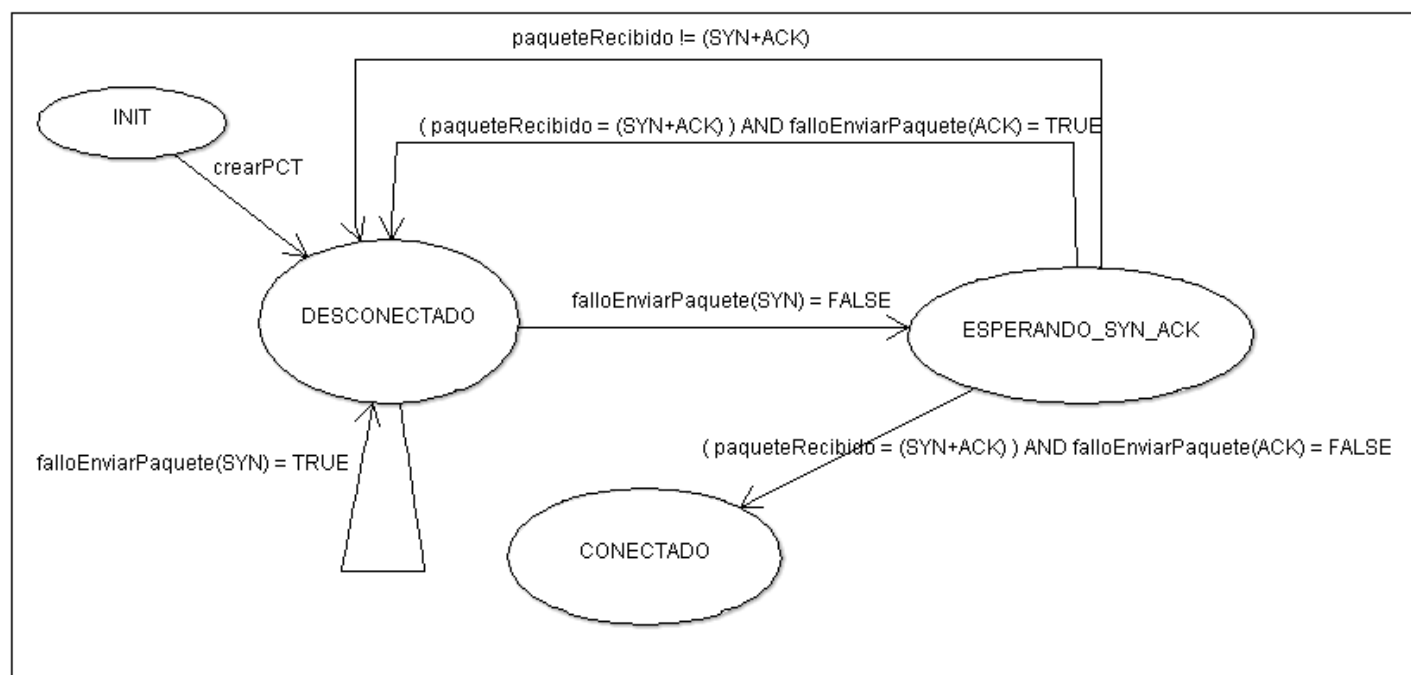
Un caso particular ocurre cuando se envía el ultimo ACK, ya que el que envía el ACK queda en estado “conectado” pero el otro lado nunca recibe el ACK por lo que no llega a conectarse. En este caso se desconectara por timeout al no recibir paquetes.

Por lo que dicho mecanismo de ARQ comenzara a ejecutarse luego de lograda la conexión.

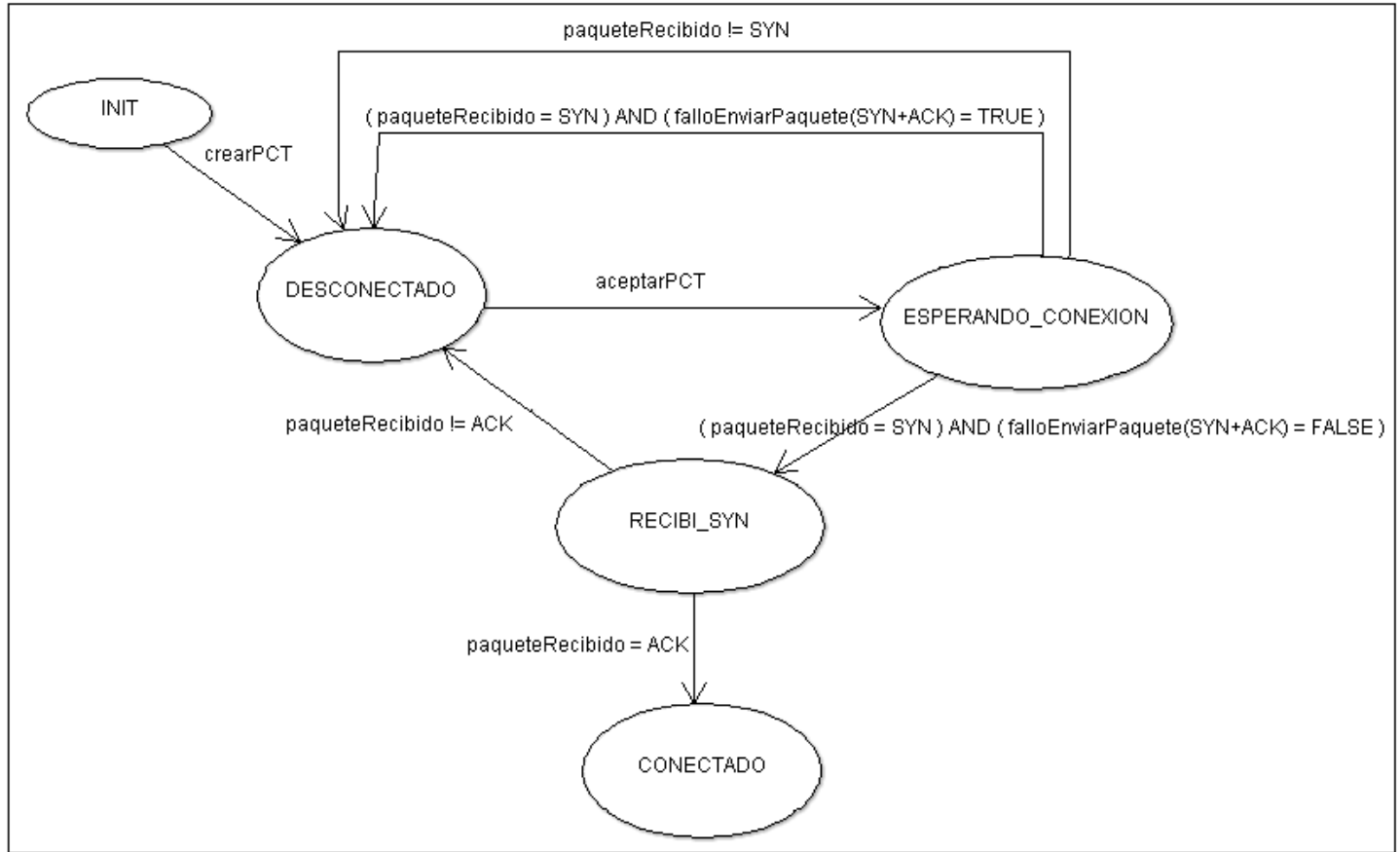
Dado que el emisor y receptor del Go Back N utilizan un número de secuencia inicial por defecto (0), no es necesario definirlo en el intercambio de paquetes del three-way handshake, por lo que se simplifica esta interacción.



A continuación se muestran las máquinas de estados donde se detalla cómo se lleva a cabo la conexión de las aplicaciones y por los estados intermedios que atraviesan. Hay que tener en cuenta que por motivos de simplicidad no mostramos en este esquema los reintentos que podrían llegar a darse.



MAQUINA DE ESTADOS DEL EMISOR



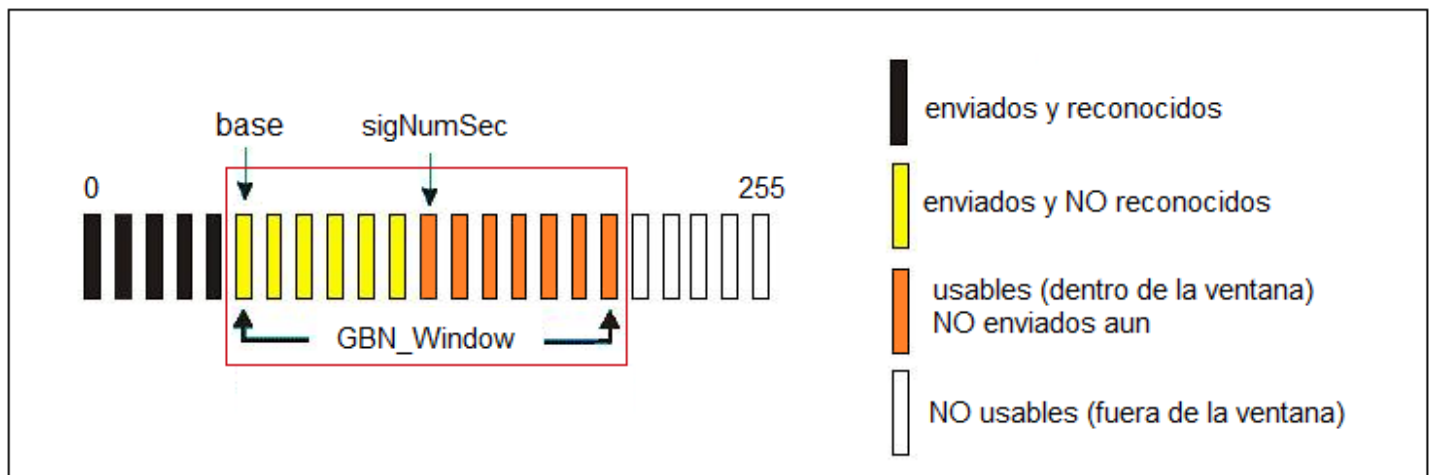
MAQUINAS DE ESTADOS DEL RECEPTOR

Descripción de Go Back N

Como ya dijimos anteriormente, para el control del flujo de datos entre emisor y receptor se implementará el protocolo Go Back N. Para su implementación se decidió encapsular todos los aspectos del funcionamiento de este protocolo en una librería.

A continuación se detallaran los aspectos relevantes sobre el funcionamiento de este protocolo y su implementación:

- Utiliza un mecanismo de ventanas en el que el emisor puede enviar los paquetes que se encuentran dentro de una “ventana” (intervalo) sin tener que esperar ningún reconocimiento para ello. Esto se conoce como pipeline de paquetes.
- La ventana del emisor tendrá un tamaño configurable GBN_WINDOW y es lo que limita a la aplicación para el control de flujo.
- La ventana va avanzando a medida que se confirman las recepciones de los paquetes enviados.
- El receptor solo acepta los paquetes en orden, o sea que no acepta un paquete hasta no haber recibido los anteriores, por lo tanto alcanza con que el tamaño de su ventana de recepción sea uno.
- El emisor debe almacenar temporalmente en un buffer hasta GBN_WINDOW paquetes mientras espera sus ACK, ya que en caso de error deberá reenviar toda la ventana. Esto ocurre debido a que el receptor sólo acepta los paquetes en orden.
- Los números de secuencia se incrementarán módulo 256, para asegurarse que pueden ser contenidos en el campo *nro_SEC_pct* y *nro_ACK_pct*, que son 8 bits por ser unsigned char, dentro del cabezal PCT.
- La ventana del emisor se implementara como un buffer circular por lo que se utilizará aritmética módulo 256.
- Los paquetes de $[0, \text{base}-1]$ corresponden a los paquetes enviados y reconocidos, los paquetes de $[\text{base}, \text{sigNumSec}-1]$ corresponden a paquetes enviados pero no reconocidos hasta el momento y los paquetes de $[\text{sigNumSec}, \text{base} + \text{GBN_WINDOW} - 1]$ se pueden emplear para el envío de nuevos paquetes.



- Con propósitos de debug se decidió agregar un pequeño fragmento de datos cuando se crean los paquetes de reconocimientos (ACK) que muestran el número de ACK y así poder visualizarlos al ejecutar el Wireshark.

Paquetes Keep Alive

La existencia de estos paquetes reside en la necesidad de la verificación de la existencia de la conexión en los momentos que no se está recibiendo ningún otro tipo de paquete ya sean datos o paquetes de control.

La implementación de la verificación se lleva a cabo mediante:

- Una rutina que se encarga de realizar dos esperas de un tiempo timeoutKAL, de forma que luego de la primera, se envía un KAL, luego se realiza la segunda espera, en caso de completarse la misma, la conexión es marcada como desconectada y con esto se indica que se detenga el envío y recepción de paquetes.
- El llamado a la rutina anterior que es realizado en los hilos de ejecución que se encargan de la recepción de paquetes, inmediatamente antes del llamado a la función que retorna el paquete recibido. Luego ante la llegada de algún paquete la ejecución de la rutina es interrumpida.

De esta forma tenemos que, si no se registra recepción de paquetes por un tiempo timeoutKAL, se realiza el envío de un paquete KAL de forma de “avisar que aún estamos activos”, y si transcurre otro tiempo timeoutKAL sin recibir paquetes se asume perdida la conexión y se detiene el envío y recepción.

Este mecanismo es adoptado tanto por el emisor como el receptor de forma de que, en caso de inactividad en el envío de datos, pero sin darse el cierre de la conexión, ambos sabrán de la existencia de la conexión debido a los KALs enviados el uno al otro.

Bibliotecas

Para la realización de la tarea modularizamos la implementación en diferentes librerías estas nos permitió una mejor distribución de trabajo entre los integrantes del grupo.

pctUtils: Una librería con funciones auxiliares que se encargue de todos los aspectos de la conexión, como el 3way handshake, la creación, envío y recepción de los distintos paquetes.

buffer: Encargado de la creación y funciones básicas del buffer circular que utilizamos.

GoBackN: Encapsulamos todos los aspectos del goBackN en esta librería

Logger: Nos permite imprimir en pantalla con distintos niveles de detalle.

Pseudocódigo

int crearPCT(struct in_addr localIPAddr)

```
{  
  
    IF (estadoActual != INIT) THEN error;  
    IF ( error al CREAR SOCKET) THEN error  
    ESTADO = DESCONECTADO;  
    SETEO GLOBAL de IP local y socket;  
    CREO estructura del buffer;  
    RETURN socket creado;  
}
```

int aceptarPCT(unsigned char localPCTport)

```
{  
  
    rol = RECEPTOR;  
    SETEO global del puerto local  
  
    IF (ESTADO != DESCONECTADO) THEN error  
  
    ESTADO = ESPERANDO_CONEXION;  
  
    WHILE (no reciba un paquete SYN y no se llevo al máximo intentos) {  
        Sleep 1  
        Intentos++  
    }  
  
    IF (intentos > MAX_INTENTOS) THEN error  
  
    ESTADO = RECIBI_SYN;  
  
    ENVIO PAQUETE (SYN ACK);  
    WHILE (no recibo paquete ACK y no se llevo al máximo de intentos) {  
        Sleep 1;  
        Intentos++  
        ENVIO PAQUETE (SYN ACK);  
    }  
  
    IF (intentos > MAX_INTENTOS) THEN error  
  
    ESTADO = CONECTADO;  
  
    INICIO receptor del GN  
  
    CREO THREAD (para leer datos y enviar al buffer)
```



```
    RETURN 0;
}
```

```
int conectarPCT(unsigned char localPCTport,unsigned char peerPCTPport, struct in_addr
peerIPaddr)
```

```
{
    rol = EMISOR;

    IF (estadoActual != DESCONECTADO) THEN error

    IF (localPCTport == peerPCTPport) THEN error
    SETEO globalmente localPCTport, peerPCTPort, peerIP

    ENVIO PAQUETE (SYN)
    ESTADO = ESPERANDO_SYN_ACK;

    int intentos = 0;
    WHILE (no recibo paquete SYN ACK y no se lleo al máximo de intentos) {
        Sleep 1;
        Intentos++;
        ENVIO PAQUETE (SYN)
    }
    IF (intentos> MAX_INTENTOS) THEN error

    ENVIAR PAQUETE (ACK)

    ESTADO = CONECTADO;

    INICIO VENTANA DEL GBN

    CREO THREAD (para leer buffer y envía datos)
    CREO THREAD ( recepción de paquetes de control)

    RETURN 0;
}
```

leerPCT:

```
Inicializo cantDatosLeidosBuffer = 0;

Si estadoConexion = DESCONECTADO
    Retorno -1
Sino
    cantDatosLeidosBuffer = readBuffer () //Le del buffer lo máximo que pueda

Retorno cantDatosLeidosBuffer;
```

escribirPCT:

```
Inicializo cantDatosEscritosBuffer = 0;

Si estadoConexion = DESCONECTADO
    Retorno -1
Sino
    cantDatosEscritosBuffer = writeBuffer () //Escribo en el buffer lo máximo que pueda

Retorno cantDatosEscritosBuffer;
```

cerrarPCT:

```
CantIntentos = 1;

Si soy EMISOR
    Si estadoActual = CONECTADO
        Mientras (No este vacio Buffer) Y (No haya llegado el ultimo ACK esperado)
            Realizo espera (30 milisegundos);

        resultadoEnvio = EnvioPaqueteControl(FIN_FLAG)

        Si resultadoEnvio = OK
            Lanzo Hilo para contar el tiempo de espera del FIN-ACK
            Mientras (estadoActual = DESCONECTADO) Y
                (cantIntentos<=cantIntentosFin)
                Si (Llego límite tiempo de espera)
                    Incremento cantIntentos
                    resultadoEnvio = EnvioPaqueteControl(FIN_FLAG)
                    Si resultadoEnvio = OK
                        Lanzo Hilo para contar el tiempo de espera del FIN-ACK;
                    Sino
                        Destruyo Buffer;
                        DestruyoEmisor //Libera memoria reservada del GBN
                        Retorno -1
                //Sale Mientras
            Si (cantIntentos > cantIntentosFin)
```

```
        Destruyo Buffer;
        DestruyoEmisor //Libera memoria reservada del GBN
        Retorno -1
    Si (estadoActual = DESCONECTADO) //Llego el FIN-ACK
        Destruyo Buffer;
        DestruyoEmisor //Libera memoria reservada del GBN
        Retorno 0

    Sino
        Destruyo Buffer;
        DestruyoEmisor //Libera memoria reservada del GBN
        Retorno -1
Sino
    Destruyo Buffer;
    DestruyoEmisor //Libera memoria reservada del GBN
    Retorno 0

Si soy RECEPTOR
    Si estadoActual = CONECTADO

        resultadoEnvio = EnvioPaqueteControl(FIN_FLAG)

        Si resultadoEnvio = OK
            Lanzo Hilo para contar el tiempo de espera del FIN-ACK
            Mientras ((estadoActual = DESCONECTADO) Y
                (cantIntentos<=cantIntentosFin))
                Si (Llego límite tiempo de espera)
                    Incremento cantIntentos
                    resultadoEnvio = EnvioPaqueteControl(FIN_FLAG)
                    Si resultadoEnvio = OK
                        Lanzo Hilo para contar el tiempo de espera del FIN-
                        ACK;
                    Sino
                        Destruyo Buffer;
                        DestruyoEmisor //Libera memoria reservada del
                        GBN
                        Retorno -1
                //Sale Mientras
                Si (cantIntentos > cantIntentosFin)
                    Destruyo Buffer;
                    DestruyoEmisor //Libera memoria reservada del GBN
                    Retorno -1
                Si (estadoActual = DESCONECTADO) //Llego el FIN-ACK
                    Destruyo Buffer;
                    DestruyoEmisor //Libera memoria reservada del GBN
                    Retorno 0

            Sino
                Destruyo Buffer;
                DestruyoEmisor //Libera memoria reservada del GBN
                Retorno -1
        Sino
            Destruyo Buffer;
            DestruyoEmisor //Libera memoria reservada del GBN
            Retorno 0
```

Preguntas

Pregunta1:

Análisis del comportamiento del protocolo PCT, frente a la variación del tamaño de ventana GBN_WINDOW del ARQ Go Back N. En especial considere el caso N=1 (ARQ Stop&Wait) y el comportamiento ante el crecimiento de N.

Es de esperar que a medida que la ventana crece la transmisión debería de ser más fluida, o sea más rápida. En particular si tomamos en cuenta que el protocolo GBN maneja un timer por ventana al haber manejado más ventanas durante el recorrido la espera para que el algoritmo solucione los errores de secuencia de paquetes sería mayor y la chace de que lleguen los paquetes esperados sería menor. Realizamos pruebas para corroborar nuestra hipótesis.

Para configuración netEmulatr

```
DELAY="100ms 10ms"  
LOSS="10%"  
REORDER="25% 50%"  
DUPLICATE="4%"
```

con envía file , 5 seg de timeout

Tamaño Ventana 1

Tiempo total de transmisión desde el primer paquete al último Medido con Wireshark 27.79 ultimo ACK
backlog 2141b 0p requeues 8)
stat Sent 29691 bytes 179 pkt (dropped 13, overlimits 0 requeues 7)

Tamaño Ventana 25

Tiempo total de transmision desde el primer paquete al último Medido con Wireshark 25.52 Seg
stat Sent 29691 bytes 179 pkt (dropped 13, overlimits 0 requeues 7)

Tamaño Ventana 50

Tiempo total de transmision desde el primer paquete al último Medido con Wireshark 25.40 Seg
Sent 31189 bytes 174 pkt (dropped 26, overlimits 0 requeues 7)

Tamaño Ventana 100

Tiempo total de transmision desde el primer paquete al último Medido con Wireshark 25.56 Seg
Sent 33241 bytes 193 pkt (dropped 20, overlimits 0 requeues 15)

Tamaño Ventana 150

Tiempo total de transmision desde el primer paquete al último Medido con Wireshark 33.49 Seg
Sent 37818 bytes 220 pkt (dropped 29, overlimits 0 requeues 25)

Modificando sleep de enviaFile

Tamaño Ventana 1

9.6 seg al ultimo ACK

Stat Sent 9412 bytes 60 pkt (dropped 7, overlimits 0 requeues 8)

Tamaño Ventana 1

5.6 seg al último ACK

Sent 21922 bytes 130 pkt (dropped 15, overlimits 0 requeues 12)

Tamaño Ventana 5

5.60 seg al último ACK

Stat Sent 34961 bytes 202 pkt (dropped 18, overlimits 0 requeues 20)

Tamaño Ventana 25

5.35 seg al último ACK

Stat Sent 34866 bytes 204 pkt (dropped 16, overlimits 0 requeues 14)

Tamaño Ventana 50

5.52 seg al último ACK

Stat Sent 37693 bytes 222 pkt (dropped 20, overlimits 0 requeues 15)

Quito sleep de enviaFile

Tamaño Ventana 1

10.74 seg al ultimo ACK

Stat Sent 10934 bytes 65 pkt (dropped 11, overlimits 0 requeues 2)

Tamaño Ventana 2

5.23 seg al último ACK

Sent 16850 bytes 102 pkt (dropped 8, overlimits 0 requeues 10)

Tamaño Ventana 5

5.73 seg al último ACK

Sent 45867 bytes 269 pkt (dropped 22, overlimits 0 requeues 11)

Tamaño Ventana 25

3.74 seg al último ACK

Sent 59870 bytes 331 pkt (dropped 49, overlimits 0 requeues 15)

Conclusiones

Si no quitamos el sleep o al menos bajamos a 1 seg su valor de EnvíaFile las diferencias de tiempos no son apreciables. Notamos que cuanto más grande es la ventana se eliminan más paquetes por el ruido (generalmente) esto puede deberse a como se decida eliminar los paquetes.

Para el caso de que el tamaño de la ventana valga 1 se tiene un retraso apreciable, a partir de 2 los tiempos se emparejan y en algunos casos aumentan independientemente del tamaño de la ventana debido a que hay más pérdida por lo indicado anteriormente.

En resumen se obtiene como conclusión que para ajustar el tamaño de la ventana del GBN se tendrían que hacer pruebas de acuerdo al contexto donde se ejecute para conseguir el mejor rendimiento. No siempre más grande es mejor. Pero sin dudas que el tener una ventana nos da una ventaja en tiempo, y en caso del GBN nos permite por ejemplo perder más ACK sin tener que reenviar paquetes puesto que la probabilidad de que se pierda un ACKs disminuye por lo que generalmente por pérdidas de ACK siempre avanzamos.

En cuanto a la pérdida de paquetes de datos baja la probabilidad de perder el paquete inicial, dado que estamos enviando más paquetes.

Pregunta2:

Indique qué cambios debería realizar al protocolo para que fuera capaz de manejar un stream de bytes (como hace TCP) en lugar de manipular paquetes.

Para poder manejar un stream de bytes deberíamos definir un número de secuencia para cada byte enviado/recibido a través de la conexión, igual que como lo hace el protocolo TCP. Quien lo reciba cuenta el número de bytes que llegaron y responde con un número de ACK = cantidad bytes + número de secuencia que llegó.

El cliente recibe y pone el número de secuencia = ack number recibido. Para ello deberíamos modificar los campos para que soporten los distintos números de ACK posibles, como también tomar en cuenta la cantidad de bytes recibidos por paquete para setear el ACK de respuesta. En el GBN tomar en cuenta la nueva numeración de ACK para el manejo de su lógica tanto en el emisor como el receptor.

Manual de ejecución

Utilizamos 3 terminales

En una terminal

```
{ruta lab2}/src/make clean
```

```
{ruta lab2}/src/make
```

```
{ruta lab2}/src/sudo ./recibeFile <puerto_orig> <direccionIP_local> <nom_archivo_salida>
```

Ejemplo `sudo ./recibeFile 1 127.0.0.1 rec_globedia.gif`

En otra terminal

```
{ruta lab2}/src/sudo ./enviaFile <puerto_local> <direccionIP_local> <puerto_remoto> <direccionIP_remoto>  
<nombre_archivo>
```

Ejemplo `{ruta lab2}/src /sudo ./enviaFile 3 127.0.0.3 1 127.0.0.1 logo_globedia.gif`

En la tercera terminal

```
sudo ./netEmulator.sh up : para emular ruido en la red
```

```
sudo ./netEmulator.sh down: para dejar de introducir ruido en la red
```

```
sudo ./netEmulator.sh stat : para chequear la estadísticas de paquetes afectados.
```

Además podríamos necesitar una cuarta terminal para el Wireshark y chequear así el tráfico en la Red

```
./sudo wireshark
```

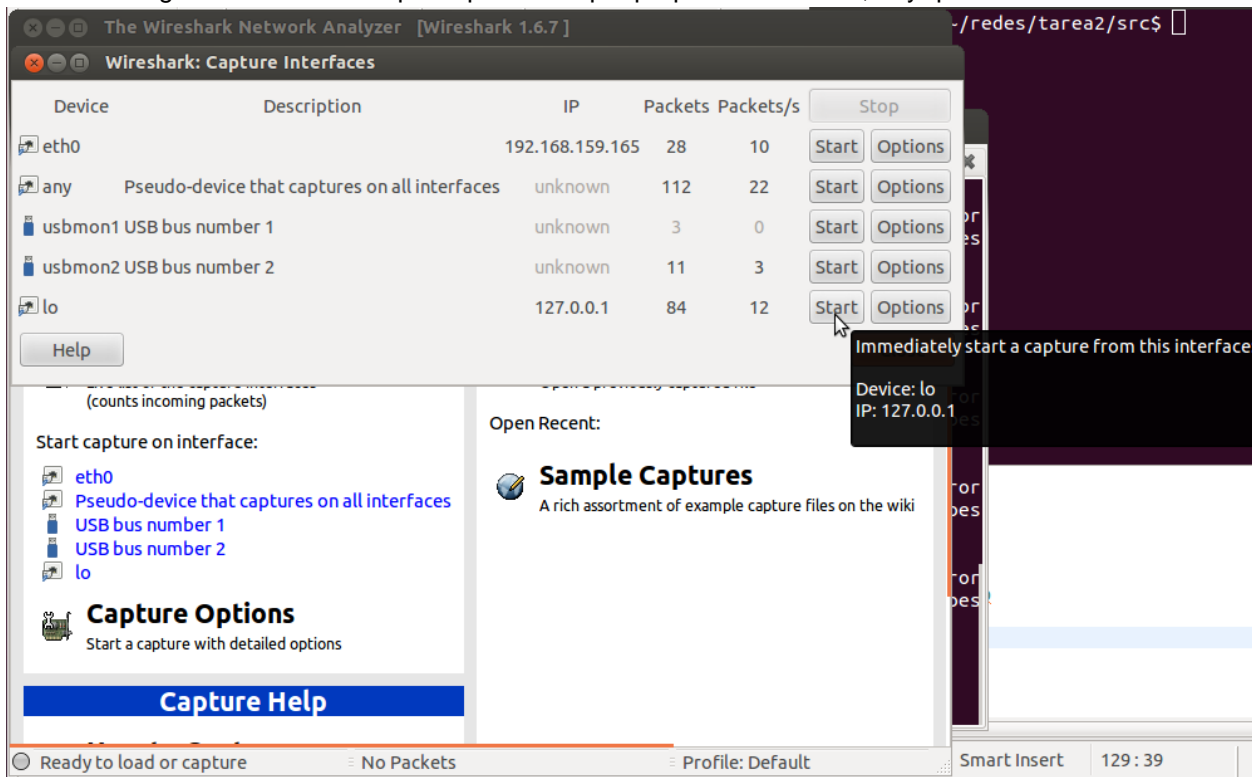
Pruebas

Para la realización de las pruebas utilizamos 3 implementaciones diferentes:

- **recibeTest y enviaTest:** fueron creadas por nosotros para el rápido testeo de nuestras soluciones, no es necesario ponerle parámetros pues estos están ya seteados en la implementación. Su funcionamiento es enviar una cadena de texto que se muestra del otro lado si llega correctamente.
- **sendFile y recibeFile:** Es parte de los insumos dados para la tarea. Estas fueron nuestra segunda tanda de pruebas. Una vez comprobado que cadenas de texto se enviaban correctamente pasamos a enviar imagenes (bytes)
- **receptor y transmisor:** Es parte de los insumos2 dados para la tarea. En esta se envían cadenas de texto, lo interesante de las mismas es que usamos el netEmulator en medio de las mismas para simular condiciones de perdida de paquetes.
- **NetEmulator:** El funcionamiento es sencillo, para prenderlo se hace `sudo ./netEmulator.sh up`, para apagarlo `sudo ./netEmulator.sh down`, para ver estadísticas y comprobar si esta prendido o apagado `sudo ./netEmulator.sh stat`. Para cambiar su configuración se puede editar el script. Editamos el script para comprobar que funcionara con distintos tipos de configuración.
- **Consideraciones importantes:** Agregamos un pequeño fragmento de datos cuando creamos los ACK y KAL (paquete keepalive) que muestra el número de ACK para propósitos de debug y para poder verlos en el wireshark esto fue pensado para facilitar la defensa. En una implementación real, los paquetes de control no tiene fragmento de datos.

Detalle de la prueba

Primero configuramos Wireshark para que intercepte paquetes correctos, hay que seleccionar el device lo:

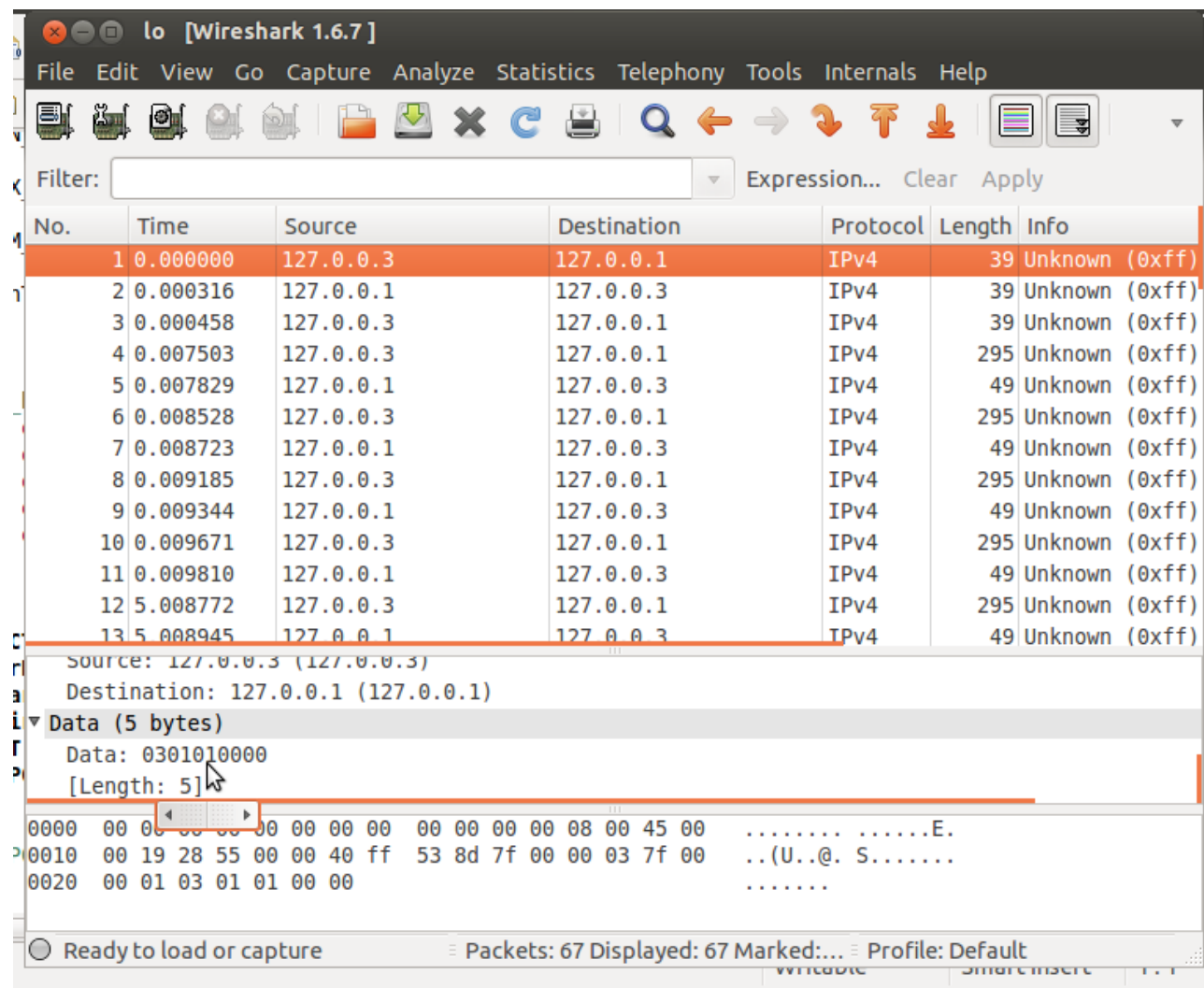


Comenzamos la recepción y transmisión de un archivo de imagen.

Recepción: `sudo ./recibeFile 1 127.0.0.1 rec_globedia.gif`

Transmisión: `sudo ./enviaFile 3 127.0.0.3 1 127.0.0.1 logo_globedia.gif`

3way HandShake, envío el SYN. Se envía desde el cliente(127.0.0.3) al servidor (127.0.0.1) . Comprobamos que es un paquete SYN pues está marcado con 01 (indicado por el puntero del mouse). Recordar que el header tcp cuenta con 5 secciones, siendo la tercera las flags: Data -> 0x80, SYN -> 0x01, ACK -> 0x02, FIN -> 0x04



3way HandShake, envío el SYN ACK. Se envía desde el servidor(127.0.0.1) al cliente(127.0.0.3) . Comprobamos que es un paquete SYN ACK pues está marcado con 03 (indicado por el puntero del mouse). Esto es SYN (02) + ACK(01) .

Wireshark 1.6.7 interface showing a packet capture. The packet list displays 13 packets, all of which are 'Unknown (0xff)'. The selected packet (No. 2) is a SYN-ACK from 127.0.0.1 to 127.0.0.3. The packet details show the data field as 0103030000, with a mouse cursor pointing to the '03' byte, which is highlighted in a red box. The packet bytes pane shows the hex data 0000 00 00 00 00 00 00 00 00 00 08 00 45 00 ...E. and 0010 00 19 b4 51 00 00 40 ff c7 90 7f 00 00 01 7f 00 ...Q..@. and 0020 00 03 01 03 03 00 00

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.3	127.0.0.1	IPv4	39	Unknown (0xff)
2	0.000316	127.0.0.1	127.0.0.3	IPv4	39	Unknown (0xff)
3	0.000458	127.0.0.3	127.0.0.1	IPv4	39	Unknown (0xff)
4	0.007503	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
5	0.007829	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
6	0.008528	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
7	0.008723	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
8	0.009185	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
9	0.009344	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
10	0.009671	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
11	0.009810	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
12	5.008772	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
13	5.008945	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)

Source: 127.0.0.1 (127.0.0.1)
Destination: 127.0.0.3 (127.0.0.3)
Data (5 bytes)
Data: 0103030000
[Length: 5]

0000 00 00 00 00 00 00 00 00 00 08 00 45 00E.
0010 00 19 b4 51 00 00 40 ff c7 90 7f 00 00 01 7f 00 ...Q..@.
0020 00 03 01 03 03 00 00

Ready to load or capture Packets: 67 Displayed: 67 Marked:... Profile: Default

3way HandShake, envío el ACK. Se envía desde el cliente (127.0.0.3) al servidor(127.0.0.1) . Comprobamos que es un paquete ACK pues está marcado con 02 (indicado por el puntero del mouse). Aquí se logra la conexión. Como se ve en el screenshot los 3 primeros paquetes son de control (Length 39).

Wireshark 1.6.7

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.3	127.0.0.1	IPv4	39	Unknown (0xff)
2	0.000316	127.0.0.1	127.0.0.3	IPv4	39	Unknown (0xff)
3	0.000458	127.0.0.3	127.0.0.1	IPv4	39	Unknown (0xff)
4	0.007503	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
5	0.007829	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
6	0.008528	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
7	0.008723	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
8	0.009185	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
9	0.009344	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
10	0.009671	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
11	0.009810	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
12	5.008772	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
13	5.008945	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)

Source: 127.0.0.3 (127.0.0.3)
Destination: 127.0.0.1 (127.0.0.1)

▼ Data (5 bytes)
Data: 0301020000
[Length: 5]

```

0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....E.
0010  00 19 28 56 00 00 40 ff 53 8c 7f 00 00 03 7f 00  ..(V..@. S.....
0020  00 01 03 01 02 00 00  .....
  
```

Ready to load or capture Packets: 67 Displayed: 67 Marked:... Profile: Default

Envío ACK desde el receptor confirmando el arribo de los paquetes. Agregamos en el paquete ACK una porción de DATA para mostrar el ACK number como se aprecia aquí se lee ACK-2 :

Wireshark 1.6.7 interface showing a packet capture. The packet list shows 13 packets. Packet 9 is selected, showing details for an IPv4 packet from 127.0.0.1 to 127.0.0.3. The data field shows a hex string 010302000241434b2d320000000000. The packet bytes pane shows the hex data with ASCII text '...ACK-2...' circled in red.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.3	127.0.0.1	IPv4	39	Unknown (0xff)
2	0.000316	127.0.0.1	127.0.0.3	IPv4	39	Unknown (0xff)
3	0.000458	127.0.0.3	127.0.0.1	IPv4	39	Unknown (0xff)
4	0.007503	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
5	0.007829	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
6	0.008528	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
7	0.008723	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
8	0.009185	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
9	0.009344	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
10	0.009671	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
11	0.009810	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
12	5.008772	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
13	5.008945	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)

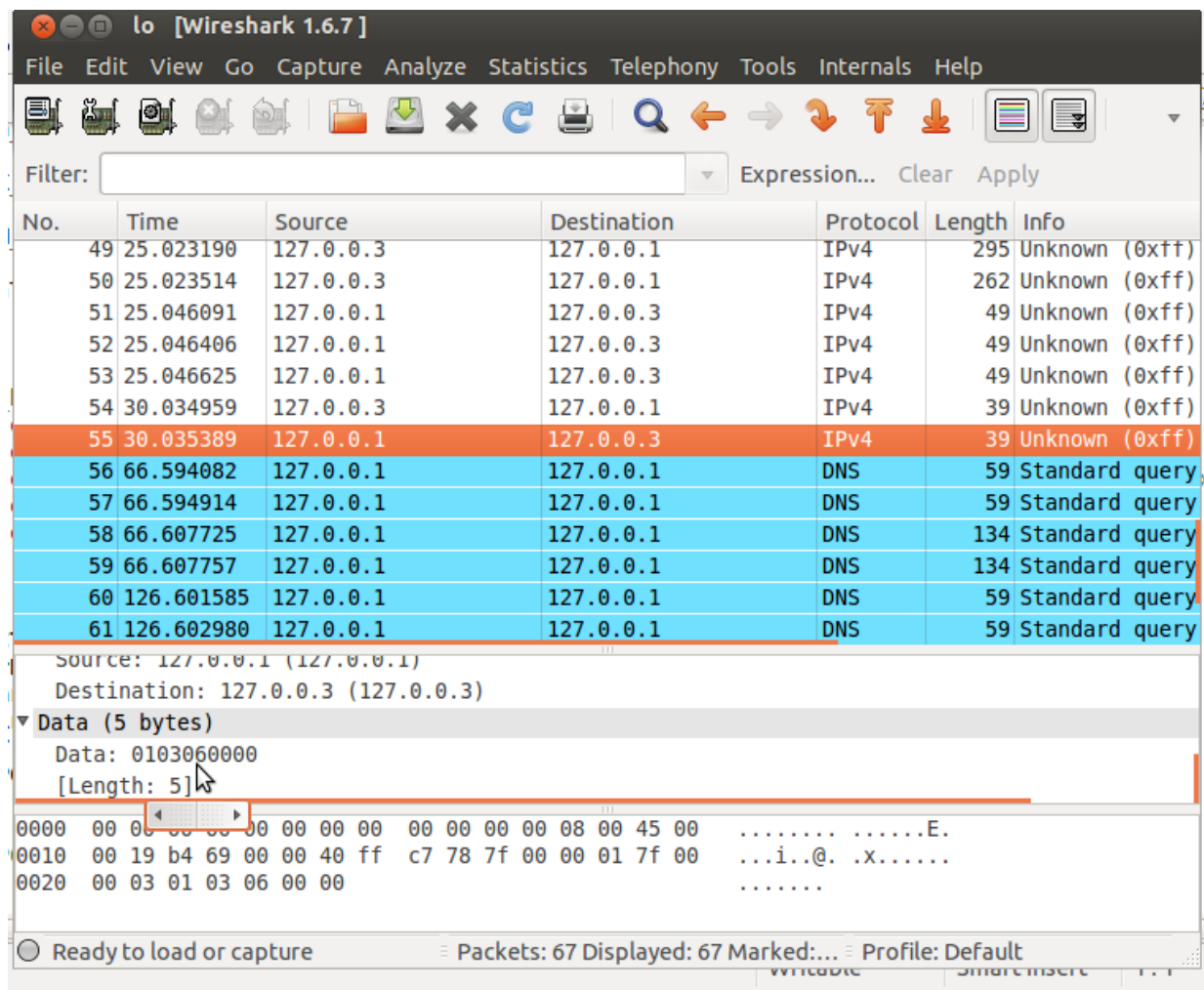
Source: 127.0.0.1 (127.0.0.1)
Destination: 127.0.0.3 (127.0.0.3)

▼ Data (15 bytes)
Data: 010302000241434b2d320000000000
[Length: 15]

0000 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00E.
0010 00 23 b4 54 00 00 40 ff c7 83 7f 00 00 01 7f 00 ...#.T.g.....
0020 00 03 01 03 02 00 02 41 43 4b 2d 32 00 00 00 00A CK-2..
0030 00

Ready to load or capture Packets: 67 Displayed: 67 Marked: ... Profile: Default

Envío de paquete FIN ACK, Se aprecia en la FLAG que es un FIN ACK 06 = (04+02)



Wireshark 1.6.7 interface showing a packet capture. The packet list shows packet 55 selected, which is an IPv4 packet from 127.0.0.1 to 127.0.0.3. The packet details pane shows the data field with the value 0103060000. The packet bytes pane shows the raw data in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Length	Info
49	25.023190	127.0.0.3	127.0.0.1	IPv4	295	Unknown (0xff)
50	25.023514	127.0.0.3	127.0.0.1	IPv4	262	Unknown (0xff)
51	25.046091	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
52	25.046406	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
53	25.046625	127.0.0.1	127.0.0.3	IPv4	49	Unknown (0xff)
54	30.034959	127.0.0.3	127.0.0.1	IPv4	39	Unknown (0xff)
55	30.035389	127.0.0.1	127.0.0.3	IPv4	39	Unknown (0xff)
56	66.594082	127.0.0.1	127.0.0.1	DNS	59	Standard query
57	66.594914	127.0.0.1	127.0.0.1	DNS	59	Standard query
58	66.607725	127.0.0.1	127.0.0.1	DNS	134	Standard query
59	66.607757	127.0.0.1	127.0.0.1	DNS	134	Standard query
60	126.601585	127.0.0.1	127.0.0.1	DNS	59	Standard query
61	126.602980	127.0.0.1	127.0.0.1	DNS	59	Standard query

Source: 127.0.0.1 (127.0.0.1)
Destination: 127.0.0.3 (127.0.0.3)

▼ Data (5 bytes)
Data: 0103060000
[Length: 5]

0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00E.
0010 00 19 b4 69 00 00 40 ff c7 78 7f 00 00 01 7f 00 ...i..@. .x.....
0020 00 03 01 03 06 00 00

Dificultades

- Debugueo de las aplicaciones. Si bien intentamos con eclipse y ciertos tutoriales no lo logramos puesto que las aplicaciones deben de ejecutarse con sudo. Por tal motivo se utiliza en algunos casos Gdb aunque no es tan amigable.
- La interpretación de los resultados del wireshark es más complicada al ser raw sockets. Por esta razón tuvimos que agregar segmentos data en los paquetes ACK para que el wireshark me muestre el número de ACK.
- La implementación del funcionamiento de los keep alive para saber si la conexión estaba activa nos tomó más tiempo de lo esperado pues teníamos que tener en cuenta los threads de los timer, mutex y sincronización de varios aspectos de la implementación.
- El cerrarSesion nos tomó más tiempo del esperado pues había varios controles que debíamos realizar y funciones a realizar antes de cerrar la sesión. Logramos resolver varios errores relacionados con la sincronización.

Posibles Mejoras

- Robustecer la implementación del 3 way handshake para que de alguna manera si se pierde el ultimo ACK se pueda seguir el proceso de conexión y no quede un extremo conectado y otro no. Esto claramente es un problema de la forma que funciona el 3 way handshake, pues luego de enviar el ACK ya considera que está conectado.
- Agregar estadísticas dentro de nuestro GBN para poder conocer número de paquetes enviados, reenviados, números de secuencia, etc. También algo más avanzado seria poder implementar una interfaz gráfica para propósitos de debugueo que muestre los paquetes enviados, la ventana y los ACK similar al applet de java de ejemplo.

Conclusiones

- En esta tarea ya contábamos con experiencia utilizando C por lo que mucho de los problemas que tuvimos en la primera tarea no se presentaron en esta.
- Las funciones que pensábamos que eran más sencillas resultaron ser complicadas pues afectaban varios lados de la implementación como son el caso de keep alives y cerrar Sesión.
- El ruido presente en la red afecta la velocidad en la que llegan los paquetes a las aplicaciones ya que el protocolo que brinda confianza tiene que estar reenviando paquetes y controlando las secuencias.
- En el Go Back N influye el tamaño de la ventana en la velocidad de transferencia pero también es importante como se implemente la aplicación que la use, en la respuesta a la pregunta 1 damos más detalles sobre esto.

