SOLUCION EXAMEN DE INTRODUCCIÓN A LAS REDES DE COMPUTADORAS.
(ref: sirc0007.doc)

25 de JULIO de 2000.

Problema 1

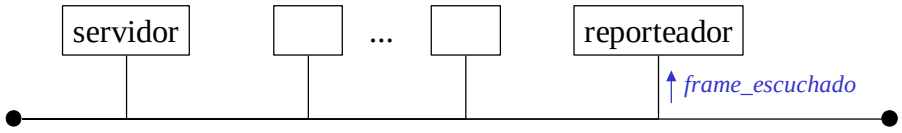
Un servidor está conectado a una red local. En un equipo conectado en la misma red local se ejecuta un programa que recibe una copia de todos los frames ethernet que son enviados en la mencionada red.

El programa recibe los frames invocando la función (bloqueante) FRAME_ESCUCHADO.

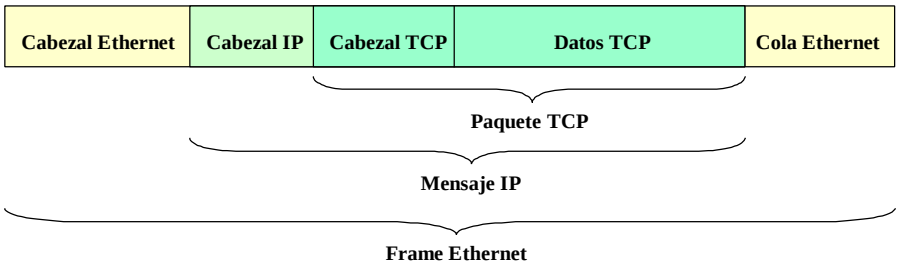
Se pide:

Adecuar dicho programa para que obtenga, para cada port TCP de la dirección del servidor para el que se detecte actividad, el promedio de paquetes IP que se emiten y reciben en cada conexión que se establece. Luego de acumular los datos de 10.000 conexiones en total, se desplegará como resultado una lista conteniendo: número de port, cantidad de conexiones ocurridas, cantidad de paquetes enviados y recibidos, promedio de paquetes por conexión.

El equipo reporteador recibe los frames ethertnet de la subred. La función frame_escuchado, devuelve el frame ethernet.



Si se trata de un paquete TCP, dentro del frame está encapsulado el paquete IP, y dentro de éste, el paquete TCP.



Se utilizarán las funciones para obtener los componentes IP y TCP:

- `Es_IP (frameETH: IN frame_ethernet, mensIP: OUT mensaje_IP): boolean;`
  Devuelve verdadero si frameETH contiene un mensaje IP, y en la variable mensIP devuelve el mismo.
  Se implementa observando el campo del paquete Ethernet que me indica que protocolo de red contiene (elijo sólo los IP).
- `Es_TCP (mensIP: IN data_IP, paqTCP: OUT paquete_TCP): boolean;`
  Devuelve verdadero si mensIP contiene un paquete TCP, y en la variable paqTCP devuelve el mismo.
  Se implementa observando el campo del paquete IP que me indica que protocolo de transporte contiene (elijo sólo los TCP).

Para establecer una conexión, el cliente envía un SYN (en la cabecera TCP) al servidor. El servidor contesta con SYN y ACK, y el cliente contesta con ACK.
Una conexión queda identificada por los campos IPorig, PORTorig, IPdest, PORTdest.
Para cerrar una conexión ambos envían un CLR.

Utilizo una tabla (variable global) para guardar estadísticas de las conexiones.

```
tabla: table of struct {
                int PORTserv,       /* puerto del servidor */
                direccionIP IPcli,  /* ip del cliente */
                int PORTcli,        /* puerto del cliente */
                int PAQcli,         /* cant paq enviados por el cliente */
                int PAQserv,        /* cant paq enviados por el servidor */
                int Estado          /* estado de la conexión */
              }
```

Se considerarán los siguientes estados:

1- Solicitud de conexión. (SYN)
2- Respuesta de conexión. (SYN-ACK)
3- Conexión establecida. (ACK)
4- Solicitud de desconexión por parte del cliente. (1er. CLR)
5- Solicitud de desconexión por parte del servidor. (1er. CLR)
6- Desconectado. (2do. CLR)

Se agregará un registro a la tabla por cada conexión que se solicite.
Se contarán los paquetes de las conexiones establecidas.
Al cerrarse una conexión no se contarán más paquetes, pero no se borrará el registro de la tabla, porque es necesario para presentar las estadísticas.
Si se conectan nuevamente se agregará otro registro.

Se utilizarán las funciones para manejar la tabla:

- `agrego_tabla (PORTserv: IN int, IPcli: IN direccionIP, PORTcli: IN int);`
  Agrega un registro con los parámetros indicados, con estado sol_cnx (1), y los contadores en 0.
  Si ya existía el registro, con conexión establecida, o a medio establecer, ignora el paquete y no agrega nada.
- `estado_tabla (PORTserv: IN int, IPcli: IN direccionIP, PORTcli: IN int): int`
  Devuelve el estado de la conexión con los parámetros indicados.
  Si no encuentra un registro para los parámetros dados, devuelve desconectado (6).
  Si hay más de un registro, sólo puede haber uno que no esté desconectado, devuelve el estado de éste, o desconectado (6) si están todos desconectados.
- `cambio_estado_tabla (PORTserv: IN int, IPcli: IN direccionIP, PORTcli: IN int, estant: IN int, estact: IN int);`
  Cambia el estado de la conexión con los parámetros indicados. El estado cambia de estant a estact.
  Si no encuentra un registro para los parámetros dados y estado estant, no hace nada.
- `sumo_cli_tabla (PORTserv: IN int, IPcli: IN direccionIP, PORTcli: IN int);`
  Suma uno al campo PAQcli, del registro con los parámetros indicados, y conexión establecida (estado=3).
  Si no encuentra un registro para los parámetros dados y estado 3, no hace nada.
- `sumo_serv_tabla (PORTserv: IN int, IPcli: IN direccionIP, PORTcli: IN int);`
  Suma uno al campo PAQserv, del registro con los parámetros indicados, y conexión establecida (estado=3).
  Si no encuentra un registro para los parámetros dados y estado 3, no hace nada.
- `vacio_tabla ();`
  Borra todos los registros de la tabla.

Se utilizará la variable global IPserv que contiene la IP del servidor.

```
int ejercicio1 () {
   frame_ethernet feth;
   mensaje_IP mip;
   paquete_TCP ptcp;
   int cnx=0;

   while (1) {
      vacio_tabla();
      while (cnx < 10000) {
         frame_escuchado(feth);
         if (es_IP(feth,mip))
            if (es_TCP(mip,ptcp))
               cuento_paquete (mip,ptcp);
      }
      estadisticas();
}
```

```
int cuento_paquete (mip: IN mensaje_IP, ptcp: IN paquete_TCP) {
   if (mip.IPdest == IPserv) {  /* paquete enviado por un cliente */
      estado= estado_tabla (ptcp.PORTdest, mip.Iporig, ptcp.PORTorig);
      case estado {
      2: if (ptcp.ACK == 1)
            cambio_estado_tabla(ptcp.PORTdest,mip.Iporig,ptcp.PORTorig,2,3);
      3: if (ptcp.CLR == 1)
            cambio_estado_tabla(ptcp.PORTdest,mip.Iporig,ptcp.PORTorig,3,4);
         else
            sumo_cli_tabla (ptcp.PORTdest, mip.Iporig, ptcp.PORTorig);
      5: if (ptcp.CLR == 1)
            cambio_estado_tabla(ptcp.PORTdest,mip.Iporig,ptcp.PORTorig,5,6);
      6: if (ptcp.SYN == 1)
            agrego_tabla (ptcp.PORTdest, mip.Iporig, ptcp.PORTorig);
      }
   }
   else if (mip.Iporig == Ipserv) { /* paquete enviado por el servidor */
      estado= estado_tabla (ptcp.PORTorig, mip.IPdest, ptcp.PORTdest);
      case estado {
      1: if ((ptcp.ACK == 1) && (ptcp.SYN))
            cambio_estado_tabla(ptcp.PORTorig, mip.IPdest, ptcp.PORTdest,1,2);
      3: if (ptcp.CLR == 1)
            cambio_estado_tabla(ptcp.PORTorig, mip.IPdest, ptcp.PORTdest,3,5);
         else
            sumo_serv_tabla (ptcp.PORTorig, mip.IPdest, ptcp.PORTdest);
      4: if (ptcp.CLR == 1)
            cambio_estado_tabla(ptcp.PORTorig, mip.IPdest, ptcp.PORTdest,4,6);
      }
   }
}

int estadisticas ()
```

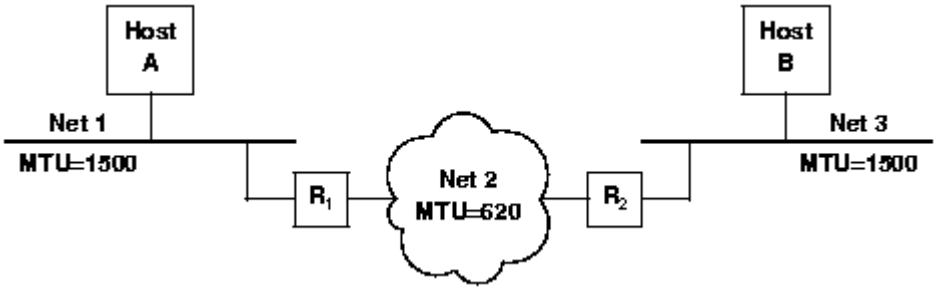
Recorre la tabla contando para cada puerto P:

- cantidad de conexiones: Cantidad de registros cuyo PORTserv = P.
- cantidad de paquetes enviados: Para los registros con PORTserv = P, totaliza el campo PAQserv.
- cantidad de paquetes recibidos: Para los registros con PORTserv = P, totaliza el campo PAQcli.
- promedio de paquetes por conexión = (cantidad de paquetes enviados + cantidad de paquetes recibidos) / cantidad de conexiones.

Luego despliega los conteos.

Problema 2

Se cuenta con dos subredes interconectadas a través de una tercera red, que intercambian datos en modo sin conexión, y con pérdida, duplicación y cambio de orden, de datagramas. El dibujo a continuación muestra un ejemplo de la situación.



Las redes Net 1 y Net 3, tienen un *tamaño máximo de transporte* (*MTU*), mayor que la red Net 2, por lo que los distintos datagramas, en caso de exceder el *MTU*, deberán ser fragmentados, y en el destino reunificados.

En el encabezado de los datagramas que maneja la red, se cuenta con los siguientes campos:

LONGITUD_TOTAL, que contiene la longitud del datagrama enviado.
IDENTIFICACION, que contiene un número diferente para cada datagrama.
RESERVA_FRAGMENTACION, (de 64 bits) que puede ser utilizado libremente.

Se pide:

Escribir el protocolo que debe correr en los ruteadores, para poder realizar la tarea de fragmentar y reunificar datagramas, suponiendo que el mismo recibe los datagramas proveniente de las subredes que necesitan ser fragmentados y los que necesitan reunificarse para ser entregados.

Considero dos canales, uno que recibe y envía los datagramas de la red con MTU menor y otro que recibe y envía los datagramas a la red con MTU mayor. Tendré las siguientes señales, rcv_mayor (recibe dtg de la red con MTU mayor), snd_mayor (envía dtg a la red con MTU mayor), rcv_menor (recibe dtg de la red con MTU menor) y snd_menor (envía dtg a la red con MTU menor).

En caso de recibir datagramas provenientes de la red con MTU mayor, los fragmento si es necesario, y los envío a la red con MTU menor. El tema de ser necesario, implica que el largo del datagrama debe exceder el MTU.

En caso de recibir datagramas provenientes de la red con MTU menor, verifico si están fragmentados, y en caso de estarlo guardo la información contenida en el datagrama, para el rearmado del mismo.

Para poder armar los fragmentos recibidos en el campo *RESERVA_FRAGMENTACION*, utilizo la siguiente estructura:

```
STRUCT RESERVA_FRAGMENTACION {
                Bit Fragmentado,        /* Indica datagrama fragmentado */
                Bit ultimo_fragmento,   /* Indica que es el último fragmento */
                long offset_fragmento   /* Indica el offset del fragmento */
        }
```

Esta información viajará en cada uno de los datagramas en el campo mencionado.

Para la fragmentación utilizaré la siguiente lógica de operación (es aplicable a los que se reciben desde el MTU mayor):

- Chequeo si el largo es mayor que el MTU menor de la otra red. Si no así lo envío tal cual lo recibí.
- En caso de MTU mayor que el de la red destino:
  - Armo los fragmentos indicando para cada uno, *fragmentado* en TRUE, *ultimo_fragmento* en FALSE para todos excepto para el último que se setea en TRUE, y *offset_fragmento* con el valor del offset del fragmento.

Para la unificación utilizaré la siguiente lógica de operación (es aplicable a los que se reciben desde el MTU menor):

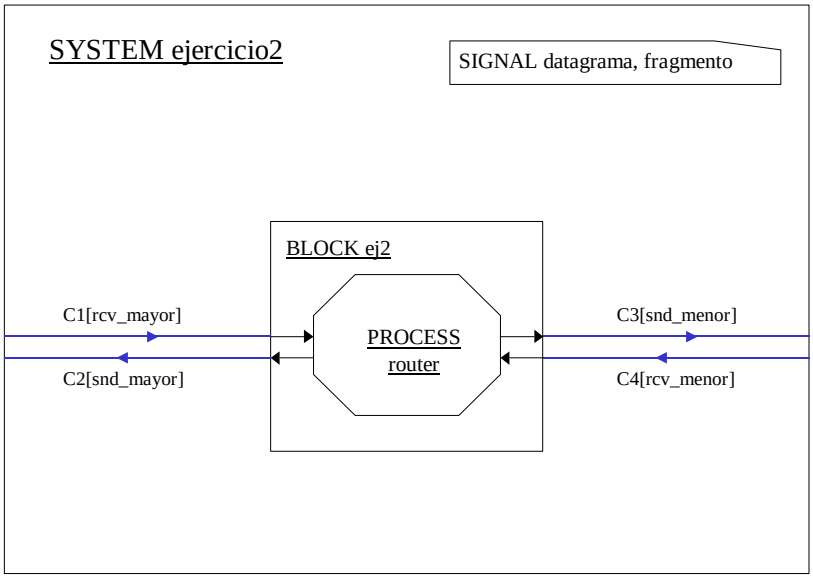
- Chequeo si *fragmentado* está en TRUE, en caso contrario lo envío tal cual lo recibí.
- En caso de *fragmentado* es TRUE:
  - Agrego el fragmento en la lista de fragmentos para el identificador del datagrama *IDENTIFICACION*, con la siguiente consideración, si el fragmento recibido es el primero para el identificador de datagrama, entonces coloco una indicación del tiempo en que lo recibí, en caso contrario solamente agrego a la lista.
  - Si *ultimo_fragmento* es TRUE, chequeo si recibí todos los fragmento del datagrama, en caso de ser así lo armo y envío a la red con MTU mayor.
- Cada un tiempo T chequeo la lista de los fragmentos, eliminando los elementos de la lista cuando la diferencia entre el tiempo actual y el tiempo de recibido, sea mayor que un TIME_OUT.

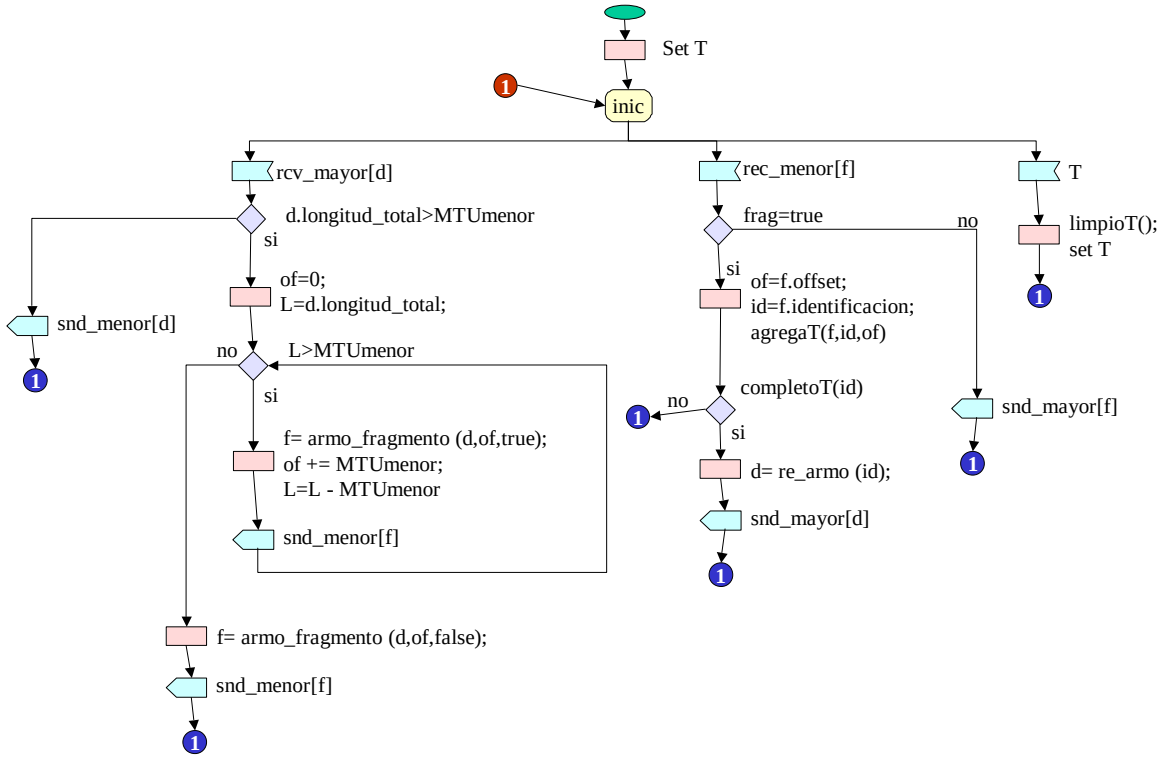
Cuando llegan fragmentos, deben reunirse todos, y luego enviar el datagrama completo a la otra red.
Se propone la siguiente estructura de datos para almacenar los fragmentos que van llegando.

```
TYPE tabla= TABLE of STRUCT {
                long id,                /* identificación del datagrama */
                lista_fragmentos frags,/* fragmentos recibidos */
                time tiempo_rcv         /* hora de recibido el primer
                                        fragmento */
            }

TYPE lista_fragmentos= LIST OF {
                long offset,            /* offset del fragmento */
                fragmento f             /* el fragmento completo */
            }

VAR T: tabla;
```

Se utilizarán las siguiente funciones para manejo de dicha estructura:

- `agregaT (f: IN fragmento, id: IN long, offset: IN long);`
  Agrega el fragmento f, a la lista de fragmentos del datagrama número id.
  Si el datagrama no estaba en la tabla (primer fragmento que llega), lo inserta en la tabla, crea una lista nueva para él, y setea el tiempo para esperar los demás fragmentos.
  Si el fragmento ya estaba en la lista (fragmento repetido), no se inserta nuevamente.
  Al insertar, se cuida el orden creciente de los offsets de fragmento.
- `completoT (id: IN long): boolean;`
  Recorre la lista de fragmentos del datagrama número id, observando si están todos los fragmentos. Como sólo se inserta una vez cada fragmento, sólo es necesario contarlos, y ver que largo_frag*cant_frag >= longitud_total. (El campo longitud total está en la cabecera de cualquiera de los fragmentos).
- `Re_armo(id: IN long): datagrama;`
  Recorre la lista de fragmentos del datagrama número id (la cual debe estar completa), concatenando los fragmentos en orden (están almacenados en orden en la lista).
  El cabezal se coloca una única vez, es decir, sólo se concatena la parte de datos de los fragmentos.
  Además borra la entrada de la tabla.
- `Armo_fragmento (d:IN datagrama,offset:IN long,ultimo:IN boolean): fragmento;`
  Devuelve la porción del datagrama comprendido entre los offset y (offset + largo_frag – 1). Si se llega al final del datagrama, el fragmento será más chico.
  Copia el cabezal del datagrama al fragmento, llenando el campo reserva_fragmentación.
- `limpioT ();`
  Vacía las entradas en la tabla para las que venció el timer.


SYSTEM ejercicio2
SIGNAL datagrama, fragmento
BLOCK ej2
PROCESS
router
C1[rcv_mayor]
C2[snd_mayor]
C3[snd_menor]
C4[rcv_menor]


Set T
inic
rcv_mayor[d]
d.longitud_total>MTUmenor
si
no
of=0;
L=d.longitud_total;
snd_menor[d]
L>MTUmenor
si
f= armo_fragmento (d,of,true);
of += MTUmenor;
L=L - MTUmenor
snd_menor[f]
f= armo_fragmento (d,of,false);
snd_menor[f]
rec_menor[f]
frag=true
no
si
of=f.offset;
id=f.identificacion;
agregaT(f,id,of)
completoT(id)
no
si
d= re_armo (id);
snd_mayor[d]
snd_mayor[f]
T
limpioT();
set T

Problema 3

Se desea implementar un servicio de transferencia de archivos que realice multiplexión descendente sobre una capa 4 orientada a conexión.

Se debe proveer la siguiente función:

> **transferir** (dirHost: direccion, cantCnx: integer, archivo: file): boolean

donde *dirHost* es la dirección del host (servidor) a donde se quiere transferir el archivo *archivo*, y *cantCnx* es la cantidad de conexiones de capa 4 que se intentarán abrir (cantCnx ≥ 1) . La función devolverá *true* si pudo transferir correctamente el archivo, y *false* en caso contrario.

El protocolo se encargará de abrir la cantidad especificada de conexiones (cantCnx) y distribuir la carga entre las mismas. Si alguna conexión no puede abrirse, o se pierde en medio de la transferencia, se continuará transfiriendo por las restantes.

La transferencia se hará en un único sentido: del cliente al servidor.
El servidor escuchará en el puerto TRANSFER.

Se cuenta con las siguientes primitivas de capa 4:

- **escuchar** (puerto: IN idPuerto)
  Deja al servidor escuchando en el puerto *puerto*, hasta recibir la conexión de un cliente.
- **conectar** (dirHost: IN direccion, puerto: IN idPuerto)
  Conecta al cliente con el servidor cuya dirección es *dirHost*, que escucha en el puerto *puerto*.
- **cerrar** (idcnx: IN idConexion)
  Cierra la conexión *idcnx*.
- **enviar** (idcnx: IN idConexion, datos: IN paquete)
  Envía el paquete *datos* por la conexión *idcnx*.

y los siguientes eventos:

- **conectado** (dirHost: OUT direccion, puerto: OUT idPuerto, idcnx: OUT idConexion)
  Se produce en ambos extremos una vez establecida la conexión. Se devuelve en *dirHost* y *puerto* la dirección y puerto del otro host, y en *idcnx* el identificador asignado a la conexión.
- **desconectado** (idcnx: OUT idConexion)
  Se produce al cerrarse o caerse una conexión, indicando la conexión en *idcnx*. Cualquier falla en el envío o recepción de paquetes producirá un evento de este tipo.
- **enviado** (idcnx: OUT idConexion)
  Indica que se produjo un envío exitoso de datos por la conexión *idcnx*.
- **recibido** (idcnx: idConexion, datos: paquete)
  Avisa de la llegada de un paquete por la conexión *idcnx*. Devuelve en *datos* el paquete que arribo.

Se pide:

> Implementar los protocolos que corren en cliente y servidor para brindar el servicio de transferencia.

La solución del problema debe resolver los siguiente puntos:

- Conservar el orden de los bloques → Se debe numerar los paquetes.
- Transmitir en paralelo por todas las conexiones, no bloquearse esperando el resultado de un envío.
- Retransmitir los paquetes ante caídas de conexión → Se debe guardar información sobre el último paquete enviado por cada conexión.
- Fragmentar adecuadamente el archivo, no dividirlo de acuerdo a la cantidad de conexiones, sino en unidades de tamaño razonable (bloques).

Las conexiones de capa 4 garantizan que no se pierdan ni desordenen los paquetes. El único error posible es la caída de la conexión, en cuyo caso el último paquete enviado puede perderse.
Sin embargo, nada nos garantiza que los paquetes que lleguen por diferentes conexiones, lleguen ordenados. Será necesario asignar un número de secuencia a los paquetes.
Se numerará a partir del número 1, reservando el paquete 0 para el envío del nombre y largo del archivo.

Para llevar la cuenta de las conexiones abiertas se utilizará la variable global CNX.

```
CNX: list of struct {
                int cid,
                int libre,
                char *dataPaq,
                int numPaq
              }
```

El campo libre indica si la conexión está oceosa (true) o está enviando datos (false). En este último caso los campos dataPaq y numPaq referencian al paquete enviado por esa conexión.
Los campos dataPaq y numPaq son necesarios en caso de caída de la conexión, para permitir el reenvío por otra conexión.

Se utilizarán las siguientes funciones para manejo de la lista de conexiones:

- `agregoCnx (int cid);`
  Agrega una conexión a la lista. La setea como libre.
- `setMsg (int cid, char *data, int num);`
  Si num>=0 setea la conexión como enviando, y registra los datos y numero de paquete. Si num=-1 pone la conexión como libre.
- `borroCnx (int cid);`
  Borra la conexión de la lista. Si la conexión estaba enviando, deja el paquete en una lista de pendientes (ver más adelante).
- `estaCnx (int cid): boolean;`
  Devuleve true si la conexión está en la lista, false sino.
- `hayCnx (): boolean;`
  Devuleve true hay alguna conexión está en la lista, false sino.
- `hayLibre (): boolean;`
  Devuleve true hay alguna conexión libre está en la lista, false sino.
- `todasLibres (): boolean;`
  Devuleve true si todas las conexiones de la lista están libres, false sino.
- `darLibre (): int;`
  Devuleve un identificador de conexión (cid), de una conexión libre.

Ni bien se recibe la indicación de establecimiento de conexión (conectado), se envían datos por esa conexión. Al llegar la confirmación de envío (enviado) se enviarán nuevos datos. En el interín se siguió enviando datos por las otras conexiones.

Si se pierde la conexión (desconectado), el paquete que se estaba enviando por la conexión que se perdió debe ponerse en una lista de paquetes pendientes. Esto lo hace la función borroCnx, ya descrita.

```
pendientes: list of struct {
                char *dataPaq,
                int numPaq
              }
```

Al principio se invoca a la función setNombre para setear el primer paquete.

- `setNombre (char *archivo);`
  Inserta en la lista un paquete con número 0, y con el nombre y largo del archivo como datos.

El archivo se lee de a bloques (read). Se enviará un nuevo bloque siempre y cuando no haya datos en la lista de pendientes.
La función darBloque se encarga de controlar eso:

- `darBloque (int &n): char*;`
  Devuelve los datos del paquete a transmitir, y por referencia el número de secuencia del mismo.
  Si hay paquetes en la lista de paquetes, devuelve un paquete de allí, y lo saca de la lista. Sino, lee un nuevo bloque del archivo, y avanza el contador de números de secuencia leídos (variable global).
  Devuelve EOF si se llegó al fin del archivo.

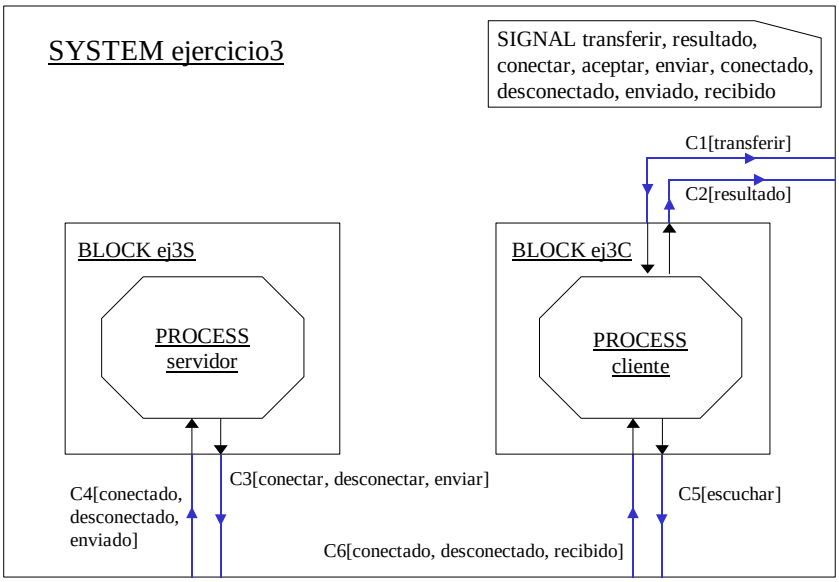
Una vez que se llegue al fin del archivo, debe esperarse la confirmación de todas las conexiones (eventos enviado). Recién cuando todas las conexiones estén libres se puede retornar el resultado *True*, además de cerrar todas las conexiones.

El servidor también tendrá una lista de conexiones abiertas, pero sólo le interesará el campo cid. El servidor no cerrará las conexiones, esperará a que el cliente las cierre, o se cierren por error.

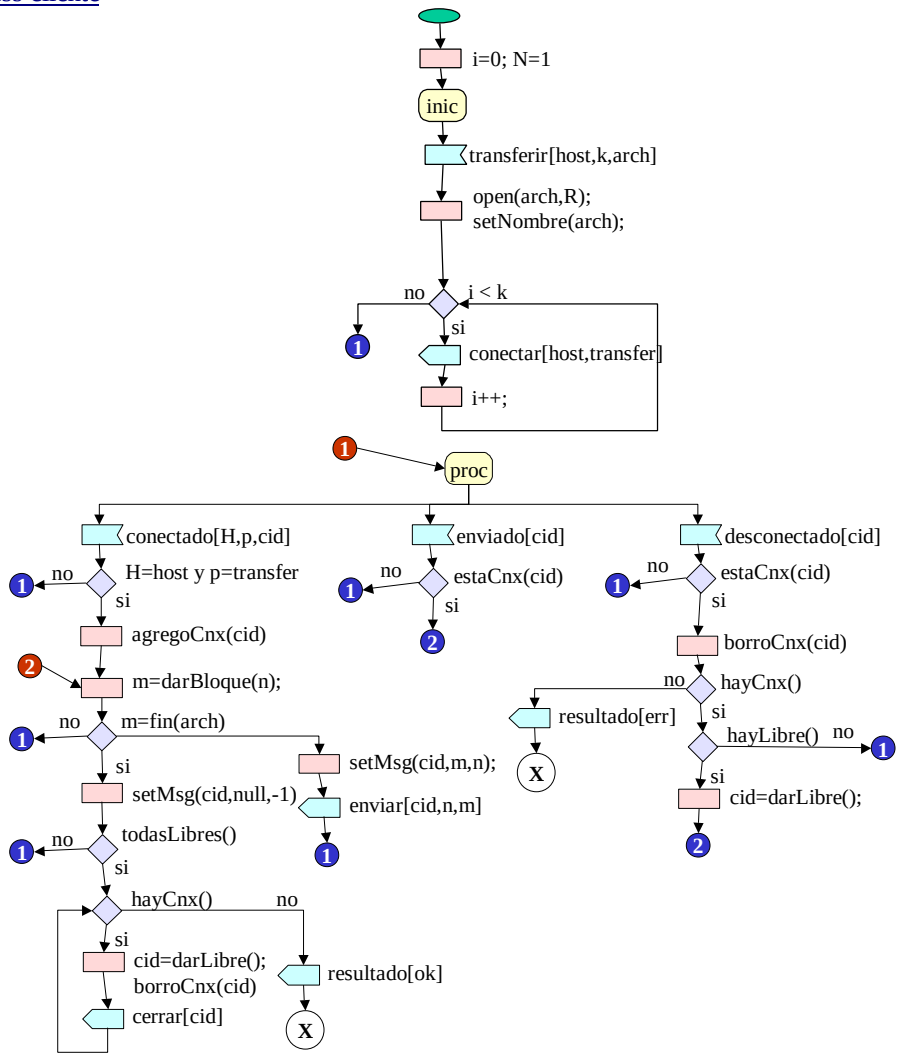
La función guardarBloque se encarga de guardar el archivo en disco (write). Debe controlar que los bloques estén en orden, por lo que mantendrá una lista de bloques fuera de orden hasta ir completando los que le faltaban. En algún sentido se comporta como ventana, pero no tiene un tamaño acotado. Con la llegada del paquete número 0, abrirá el archivo para escritura (open), y con la llegada del último paquete lo cerrará (close)

- `guardarBloque (int &n, char *data);`
  Escribe en disco los bloques, conservando su orden.

Por último en los paquetes de datos (primitiva enviar), se enviará el número de secuencia y el bloque. La cantidad de bytes destinada al número de secuencia estará delimitada por el tamaño en bloques del archivo, y es transmitida al servidor en el primer paquete.



Process cliente



Process servidor