CAPÍTULO 8 : ANEXOS

8.1. CLASES MODIFICADAS DEL SIMULADOR BÁSICO

Aquellos métodos o variables que han sido añadidos al código original están resaltados con letra en negrita para facilitar su localización.

8.1.1. SOURCE

SOURCE.H

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t
-*- */

#ifndef _source_h_
#define _source_h_

////////////// Includes del sistema


////////////// Includes de la aplicación
#include "module.h"
#include "distrib.h"
#include "lstat.h"

////////////// Defines del módulo



class Source : public Module
{
public:
                        Source        (Distrib * tll, Distrib * tsv);
  virtual               ~Source       ();
          void          start         (double tinit);
          void          stop          (double tstop);
          void          upTarget      (Module *)        { }
          void          startSource   (double now, Packet *);
          void          stopSource    (double now, Packet *);

protected:
          void          nextArrival   (double now, Packet *);
private:
  virtual void          fillPacket    (double now, Packet *p);
  virtual bool          isfull        (double now, Packet *p);
  virtual void          putPsh        (double now, Packet *p);
          bool          stopped_;
          DistStat      tLleg_;
          DistStat      tPqte_;
};

#endif
```

SOURCE.CC

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t
-*- */

#include "source.h"


Source::Source(Distrib * tll, Distrib * tpq) :
  Module(1000),
  stopped_(true),
  tLleg_(tll),
  tPqte_(tpq)
{
  connId_ = id();
}


Source::~Source()
{
}


void
Source::start(double tinit)
{
  wakeUp((void (Module::*)(double, Packet *))&Source::startSource,
tinit);
}


void
Source::stop(double tstop)
{
  wakeUp((void (Module::*)(double, Packet *))&Source::stopSource,
tstop);
}


void
Source::startSource(double now, Packet *)
{
  stopped_ = false;
  wakeUp((void (Module::*)(double, Packet *))&Source::nextArrival,
             now + tLleg_.pickValue());
}


void
Source::stopSource(double, Packet *)
{
  stopped_ = true;
}


void
```

```
Source::nextArrival(double now, Packet *)
{
  if (! stopped_ )
      {
        Packet * paquete = new Packet((int) tPqte_.pickValue());
        putPsh(now,paquete);
        if (isfull(now,paquete))
            {
              paquete->setOrigen(this);
              paquete->toDown();
              paquete->setConnId(connId_);
              fillPacket(now,paquete);
              send(now, paquete);
            }
        else //No lo utiliza->liberamos memoria dinámica.
            {
              delete (paquete);
            }
        wakeUp((void (Module::*)(double, Packet
*))&Source::nextArrival,
                    now + tLleg_.pickValue());
      }
}


bool
Source::isfull(double now, Packet *p)
{
  return true;
}


void
Source::fillPacket(double now, Packet *p)
{
      p->PacketIP::datos.uno = 1;
}

void
Source::putPsh(double now, Packet *p)
{
  p->PacketTCP::datos.psh=false;
}
```

8.1.2. PACKET

PACKET.H

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t
-*- */

#ifndef _packet_h_
#define _packet_h_
```

```
#include <stdio.h>

#define ATMCELLSIZE 53

////////////// Includes del sistema
typedef unsigned long int SIZE_;

class PacketIP
{
public:
  struct
  {
      int uno;
      int dos;
  } datos;
};

////////////////////////////////////////////////////////////
//Segmento que es transmitido entre TCPs. Cada campo indica:
//    error_header -> TRUE si al comprobar el checksum hay
//                    error en la cabecera.
//    data_size    -> tamaño en octetos de los datos de usuario.
//    seq_number   -> numero de secuencia del primer octeto de
//                    datos ( numero de serie ).
//    window       -> numero de octetos de datos empezando con
//                    el indicado en el campo ack_number que el
//                    transmisor del segmento esta dispuesto a
aceptar.
//    newWnd       -> nuevo valor de la ventana de recepción para el
extremo receptor del segmento.
//    ack_number   -> numero de asentimiento.
//    abort     -> TRUE si hay que abortar la conexión.
//    ack       -> TRUE si es significativo el campo de asentimiento.
//    psh       -> TRUE si se señala la funcion push.
//    rst       -> TRUE para resetear la conexion.
//    syn       -> TRUE sincroniza la secuencia de numeros.
//    fin       -> TRUE cuando no hay mas datos del que envia.
//    wndChange -> TRUE si se ha producido cambio de valor en la
ventana de recepción del extremo receptor del segmento.
////////////////////////////////////////////////////////////
class PacketTCP
{
public:
  struct
  {
      bool error_header;
      SIZE_ data_size;
      SIZE_ seq_number;
      SIZE_ ack_number;
      SIZE_ window;
      SIZE_ newWnd;
      bool abort;
      bool ack;
      bool psh;
      bool rst;
      bool syn;
      bool fin;
```

```
      bool wndChange;
  } datos;
};
class ATMCell
{
public:
  struct
  {
      bool taggedcell_;
      bool lastcell_;
      int  numOrden_;
  } atmCell;
};

////////////// Includes de la aplicación
class Module;
class Packet : public PacketIP, public PacketTCP, public ATMCell
{
public:
                    Packet    (double size)   { size_ = (SIZE_) size; }
         bool       isUp      ()              { return UP ==
direction_; }
         void       toUp      ()              { direction_ = UP; }
         void       toDown    ()              { direction_ = DOWN; }
         void       setOrigen (Module * from) { origen_ = from; }
         Module *   origen    ()              { return origen_; }
         void       setSize   (double size)   { size_ = (SIZE_) size; }
         SIZE_      size      ()              { return size_; }
         void       setConnId (int connId)    { connId_ = connId; }
         int        connId    ()              { return connId_; }
         double     data      ()              { return data_; }
private:
  enum       {UP, DOWN} direction_;
         Module *   origen_;
         SIZE_      size_;
         int        connId_;
         double     data_;
         double     timestamp_;
};

#endif
```

8.1.3. TIMER

TIMER.H

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:2; indent-tabs-mode:t
-*- */

#ifndef __timer_h__
#define __timer_h__

using namespace std;
#include <stack>
```

```
#include "packet.h"
#include "module.h"


/*************************************************************
 *
 * Clase genérica de temporizadores
 *
 *************************************************************/
/*************************************************************
 * Problemas a resolver:
 *   - al destruir el temporizador, puede estar pendiente
 *     alguna notificación.
 *   - ¿Avisar cuando se activa un temporizador activo?
 *   - ¿Avisar cuando se desactiva un temporizador desactivado?
 *************************************************************/
class Timer
{
public:
                                Timer           ();
                                Timer           (Module * const module,
                                                 double interval,
                              void (Module::* handler)(double, Packet *),
                                                   Packet * packet);
                                    ~Timer          ();
                    void *          operator new    (size_t t);
                    void            operator delete (void * p, size_t);
      inline        bool            operator ()     (Timer * left,
                                                     Timer * right);
      inline        double          time        () { return time_; }
      inline        Packet *        packet      () { return packet_; }
                    void            timeOut     ();
                    void            deactivate  ();
protected:
                    bool            active_;
                    Module *        module_;
                    void (Module::* wakeUp_)(double, Packet *);
                    Packet *        packet_;
                    double          time_;
private:
      static        stack<Timer *>  recycled_;
};


bool
Timer::operator () (Timer * left, Timer * right)
{
      return left->time() > right->time();
}

#endif
```

TIMER.CC

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:2; indent-tabs-mode:t
-*- */

#include "timer.h"
#include "scheduler.h"

/************************************************************
 *
 * Clase genérica de temporizadores
 *
 ************************************************************/
/************************************************************
 * Problemas a resolver:
 *   - al destruir el temporizador, puede estar pendiente
 *     alguna notificación.
 *   - ¿Avisar cuando se activa un temporizador activo?
 *   - ¿Avisar cuando se desactiva un temporizador desactivado?
 ************************************************************/
stack <Timer *> Timer::recycled_;


Timer::Timer() :
      active_(false), module_(0), wakeUp_(0), packet_(0), time_(0)
{
}

Timer::Timer(Module * const module, double interval,
      void (Module::* handler)(double, Packet *),Packet * packet) :
      active_(true), module_(module), wakeUp_(handler),packet_(packet),
      time_(interval)
{
}

Timer::~Timer()
{
      if (active_)
            deactivate();
}


void *
Timer::operator new(size_t t)
{
      Timer * newTimer;
      if (recycled_.empty())
            newTimer = (Timer *) malloc(t);
      else
            {
                  newTimer = recycled_.top();
                  recycled_.pop();
            }
      return newTimer;
}
```

```
void
Timer::operator delete(void *p, size_t)
{
      Timer * aux = (Timer *) p;
      aux->packet_ = 0;
      recycled_.push(aux);
}


void
Timer::timeOut()
{
      if (active_)
            {
                  active_ = false;
                  (module_->*wakeUp_)(Scheduler::instance()->now(),
packet_);
            }
}


void
Timer::deactivate()
{
      if (active_) {
            active_ = false;
            if (0 != packet_)
                  {
                        delete packet_;
                        packet_ = 0;
                  }
      }
}
```

8.2. CLASES DEL SIMULADOR DEL ESTÁNDAR TCP

A continuación se expone el código completo de las clases que forman parte del simulador del estándar TCP.

8.2.1. MAIN

MAIN

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

#include "../scheduler.h"
#include "../node.h"
#include "../sourceTCP.h"
#include "../sinkTCP.h"
```

```
#include "../packet.h"
#include "../media.h"



int main(int argc, char * argv[])
{
  Scheduler simInst;

  if (10 != argc && 9 != argc && 12 != argc && 11 != argc)
      {
        printf("Usage: %s <sem> <dur> <tll> <tser> <serv> <w_src> <
w_snk> [ <q> ] [ <ts_src> ] [ <ts_snk> ]\n", argv[0]);
        printf("\t<sem>    semilla\n");
        printf("\t<dur>    duración de la simulación\n");
        printf("\t<m>      parámetro m de la distribución de
Pareto\n");
        printf("\t<alfa>   parámetro alfa de la distribución de
Pareto\n");
        printf("\t<tam>    tamaño de los paquetes\n");
        printf("\t<rc>     régimen en células por tiempo de los
enlaces\n");
        printf("\t<w_src>  tamaño de la ventana para la fuente de
datos\n");
        printf("\t<w_snk>  tamaño de la ventana para el destino de los
datos\n");
        printf("\t<q>      tamaño de cola de entrada. Si ausente,
infinita\n");
        printf("\t<ts_src> tamaño del segmento de datos en la fuente.Si
ausente,8500 octetos\n");
        printf("\t<ts_snk> tamaño del segmento de datos en el
destino.Si ausente,5000 octetos\n");
        printf("\n -- EN CASO DE FIJAR SÓLO UN TAMAÑO PARA LOS
SEGMENTOS DE DATOS, ÉSTE SE CONSIDERARÁ POR DEFECTO EL DE LA FUENTE --
\n\n");
      }

  else if ((atoi(argv[8])==0))
      printf("\n\t **ERROR** :El tamaño de ventana para sink no puede
ser cero para permitir el envío de datos por parte de la fuente.\n\n");

  else if ((!(argc<11)) && (atoi(argv[10])==0))
        printf("\n\t **ERROR** : El tamaño del segmento para source no
puede ser cero para permitir el envío de datos por parte de la
fuente.\n\n");

  else
      {
        if (11 == argc)
            printf("\t **ATENCIÓN**: El simulador considera por defecto
que el valor pasado como longitud de segmento es en la fuente. En caso
de querer fijar el del destino, debe dar valor para ambos extremos,
siendo el primer valor la longitud para la fuente y el segundo el valor
para el destino.\n\n");
        for ( int i = 0 ; i < argc ; i++ )
            printf("%s ", argv[i]);
```

```
        printf("\n");

        Distrib::setSeed(atoi(argv[1]));

        SourceTCP    fuente1(new Pareto(1/atof(argv[3]),atof(argv[4])),
                  new Constant(atof(argv[5])), atoi(argv[7]),
                  (!(argc < 11)) ? atoi (argv[10]) : 8500);

        Node         nodo1(atof(argv[6]),
                     new Queue((!(argc < 10)) ? atoi(argv[9]) : -1), 1);

        Node         nodo2(atof(argv[6]),
                     new Queue((!(argc < 10)) ? atoi(argv[9]) : -1), 1);

        SinkTCP      sink1(atof(argv[6]),
                     new Queue((!(argc < 10)) ? atoi(argv[9]) : -1),
                      atoi(argv[8]),1, ( 12 == argc ) ? atoi (argv[11])
                                                          : 5000);
        Media        medio(atof(argv[6]));

        fuente1 >> nodo1 >> medio << nodo2 << sink1;
        sink1 >> nodo2 >> medio << nodo1 << fuente1;

        fuente1.start(0.0);
        fuente1.stop(atof(argv[2]));

        simInst.run();

        cout << "Nodo1: recibidos " <<
nodo1.getStatistics(Node::PktRcvd) <<
            " perdidos " << nodo1.getStatistics(Node::PktDrop) << endl;
        cout << "Sink1: recibidos " <<
sink1.getStatistics(Node::PktRcvd) <<
            " perdidos " << sink1.getStatistics(Node::PktDrop) << endl;
      }

  return 0;
}
```

8.2.2. SOURCE_TCP

SOURCE_TCP.H

```
/* -*-  Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t -*-
*/

#ifndef _sourceTCP_h_
#define _sourceTCP_h_

//////////Includes del sistema
#include <set>
#include <list>
#include <iostream>
```

```
//////////Includes de la aplicación
#include "source.h"
#include "module.h"
#include "distrib.h"
#include "lstat.h"
#include "connection.h"
#include "packet.h"
#include "timer.h"

//////////Defines del módulo
#define  MSL 2        //Maximum Segment Life
#define CONTADOR 1.5  //Marca un umbral para el contador de
retransmisiones.

class SourceTCP : public Source
{
public:
  SourceTCP  ( Distrib * tll, Distrib * tsv, int win_source , int
segmentsize );
  virtual         ~SourceTCP ();

//Sobrecarga del método start de la clase Source para simular el
establecimiento de llamada.
  void            start      (double tinit);

//Sobrecarga del método stop de la clase Source para simular la
liberación de llamada.
  void            stop       (double tstop);

private:

//Método que procesa los datos recibidos actuando en consecuencia.
  virtual  void   recvDown       (double now, Packet *p);

//Método sobrecargado de source mediante el cual se da valor a los
distintos campos del paquete TCP.
  virtual  void   fillPacket     (double now, Packet *p);

//Introduce los datos de usuario en el segmento hasta que éstos lleguen
al tamaño de segmento, completen la ventana de transmisión o se indique
desde capas superiores una función push.
  virtual  bool   isfull         (double now, Packet *p);

//Método que reenvía un determinado segmento en caso de que haya
expirado el contador de retransmisión.
  void   retxon         (double now, Packet *paq);

//Cuando llega un asentimiento válido elimina los segmentos asentidos
de la cola de retransmisión. También elimina todos los elementos
almacenados cuando el extremo se encuentra en la fase de liberación de
la conexión.
  void   TimeOut        (double now, Packet * p, bool liberacion);

//Método que inicia la liberación de llamada enviando un segmento con
el bit fin activo.
  void   stopSourceTCP  (double tstop);
```

```
//Establece estado CLOSED a partir de TIME_WAIT pasados 2MSL.
  void   stateClose     (double now, Packet *p);


//Instancia de la clase ConnectionTCP->se encarga de los parámetros de
la conexión.
  Connection    * connIns_;

//flag que me controla la primera vez que la fuente entra en estado
ESTABLISHED, es entonces cuando comenzamos el envío de datos.
  bool            firstsend_;

//Indica en cada instante el tamaño del segmento, y controla el momento
de comenzar el envío cuando el segmento está lleno.
  unsigned long int  segsize_;

//Número de paquetes recibidos.
  Counter         packetsRecv_;

//Lista de timers de retransmisión. Se activa un timer y se incluye en
la lista cada vez que se produce el envío de un segmento de datos.
  set  <Timer *>    timerTCP_;

//Al recibir un segmento válido en recepción informa a la fuente si
estaba parado el envío por quedarnos sin ventana, para que pueda
proceder a su activación.
  bool            stopSend_;

//Si estamos en un proceso de recuperación de ventana tras un estado de
congestión y llega un segmento, se va recuperando ventana y debemos
indicárselo al otro extremo. Se necesita una variable global al ser
completamente independientes la transmisión y la recepción.
  bool            newWnd_;
};

#endif
```

SOURCE_TCP.CC

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t
-*- */

#include "sourceTCP.h"
#include "scheduler.h"




SourceTCP::SourceTCP(Distrib * tll, Distrib * tpq, int win_source, int
segmentsize ) :
  Source(tll,tpq),
  segsize_(0),
  newWnd_(false),
  stopSend_(false),
  firstsend_ (true)
```

```
{
  /* Establecemos el primer número de secuencia de la fuente */
  Distrib * issnum=new Exponential (1500);
  DistStat  issn=issnum;
  connIns_=new
Connection((int)issn.pickValue(),win_source,segmentsize);
}

SourceTCP::~SourceTCP()
{
}

////////////////////////////////////////////////////////////
//
// Método: start
//
//  Función: Método start sobrecargado de la clase source con el
//objetivo de simular una llamada de usuario OPEN activa antes de
//empezar el envío de datos.
//
////////////////////////////////////////////////////////////
void
SourceTCP::start (double tinit)
{
  Packet * p=connIns_->activeOPEN();
  p->setOrigen(this);
  p->setConnId(connId_);
  send(tinit,p);
}

////////////////////////////////////////////////////////////
//
// Método: recvDown
//
// Función: Método que permite la recepción de paquetes en la fuente.
//
// Parámetros:
//    now -> instante simulado actual.
//    p   -> paquete recibido.
//
////////////////////////////////////////////////////////////
void
SourceTCP::recvDown (double now, Packet *p)
{
  ++packetsRecv_;

//En primer lugar se comprueba si el segmento entrante contiene
//errores en la cabecera como resultado de la comprobación del
//checksum, si es así se ignora.
  if (p->PacketTCP::datos.error_header==false)
      {

        //A continuación se verifica que no se nos ha indicado desde el
//otro extremo que abortemos la conexión. En caso afirmativo se detiene
//inmediatamente el envío de datos.
        if (p->PacketTCP::datos.abort)
```

```
            wakeUp((void (Module::*)(double,Packet*))
                              &Source::stopSource,now);

        else if (p->PacketTCP::datos.rst)
            {
              connIns_->reset();
              if (connIns_->abort())
                  //Si llega un mensaje especial reset durante un
estado sincronizado de la conexión hay que abortar la conexión.
                  wakeUp((void (Module::*)(double,Packet
*))&Source::stopSource,now);

            } else {
              //Antes de nada comprobamos si ha habido algún cambio en
//la ventana de recepción originado en el otro extremo a causa de un
//estado de congestión de la red. Debemos comprobarlo antes de procesar
//el segmento de entrada, puesto que la ventana me marca el límite para
//el número de secuencia del segmento entrante.
              if (p->PacketTCP::datos.wndChange)
                    connIns_->putWindow(p->PacketTCP::datos.newWnd);

              //PackProcessing es la máquina de estados de la conexión,
//su misión es comprobar el segmento de entrada y si éste no es
//inaceptable, actualizar el estado de la conexión y las variables
//asociadas a los espacios de números de secuencia y asentimiento, así
//como elaborar el segmento a enviar al otro extremo como respuesta al
//recibido.

              Packet * pack=connIns_->PackProcessing(p);
              if (!(pack==0))
                  {
                    if (connIns_->isconnEstab())
                        // En este momento tenemos la llamada
establecida.
                        {
                          //Recalculamos los parámetros de la ventana y
//tamaño de segmento si es la primera vez que entramos en estado
//ESTABLISHED, o si la red entra en un estado de congestion ( se
//produce una retransmisión debido al vencimiento de un contador de
//retransmisión ) o la ventana se está recuperando después de un estado
//de congestión.
                         TimeOut(now,p,false);
                         newWnd_=connIns_->winCalculate(firstsend_,false);
                         if (firstsend_)
                             {
                               //Primero   mandamos   el   asentimiento   al
//segmento syn recibido.
                               pack->setOrigen(this);
                               pack->setConnId(connId_);
                               pack->PacketTCP::datos.ack_number=connIns_-
>putAck();
                               send(now,pack);
                               //Luego,  como  tenemos  ya  establecida  la
//conexión, damos comienzo al envío de datos.
                               Source::start(now);
                               firstsend_=false;
                             }
```

```
                    else
                        {
                          //Al recibir un asentimiento se comprueba si
//la fuente de datos tiene parado el envío al haberse quedado sin
//ventana, en cuyo caso se reactiva.
                          if (stopSend_)
                              {
                                 wakeUp((void (Module::*)
                              (double,Packet *))&Source::startSource,
                                                      now);
                                 stopSend_=false;
                              }
                         }
                        }
                  else
                      {
                        //   En este caso estamos en fase de
//establecimiento o liberación de llamada -> intercambiamos los bits de
//control necesarios que ya hemos puesto en el msg en la llamada a
//PackProcessing.

                          //En el caso de que la fuente se encuentre en
//estado Time-Wait debe esperar 2 MSL antes de cerrar definitivamente
//la conexión según especificación del estándar.
                          if (connIns_->isconnTmWait())
                              wakeUp((void (Module::*)
                            (double, Packet *))&SourceTCP::stateClose,
                                       MSL+2);

                          pack->setOrigen(this);
                          pack->setConnId(connId_);
                          pack->PacketTCP::datos.ack_number=connIns_-
>putAck();
                          send(now,pack);
                        }
                } else if (connIns_->returnError())
                  {
                      Packet * error=connIns_->errorMsg(p,false);
                      error->setOrigen(this);
                      error->setConnId(connId_);
                      send(now,error);
                  }
            }
      }
}



//////////////////////////////////////////////////////////////
//
// Método: fillPacket
//
// Función: Establecer el valor de cada uno de los campos del paquete
// TCP
//////////////////////////////////////////////////////////////
```

```
void
SourceTCP::fillPacket(double now, Packet *p)
{
  p->PacketTCP::datos.error_header = false;
  p->PacketTCP::datos.abort=false;
  p->PacketTCP::datos.syn = false;
  p->PacketTCP::datos.ack = true;
  p->PacketTCP::datos.fin = false;
  p->PacketTCP::datos.rst = false;
  if (newWnd_)
      {
        p->PacketTCP::datos.wndChange=true;
        p->PacketTCP::datos.newWnd=connIns_->TxonWindow();
        newWnd_=false;
      }
  else
      p->PacketTCP::datos.wndChange = false;

  //Ponemos número de secuencia y de asentimiento.
  connIns_->seqNumProcessing(p);
  p->PacketTCP::datos.ack_number=connIns_->putAck();

  //Creamos un nuevo elemento Timer que se insertará en la lista de
//retransmisión.
  Timer * t_aux=new Timer(this,now+(double)CONTADOR,(void
(Module::*)(double,Packet *))&SourceTCP::retxon,p);

  //Definimos una instancia de Scheduler que encole en la lista de
eventos pendientes la posible retransmisión.
  (Scheduler::instance())->wakeUp(t_aux);

  //Insetamos el elemento en la lista.
  timerTCP_.insert(t_aux);
}


////////////////////////////////////////////////////////////
//
// Método: retxon
//
//  Función: Retransmitir un determinado segmento si ha expirado el
//contador de retransmisión.
//
////////////////////////////////////////////////////////////
void
SourceTCP::retxon(double now, Packet *p)
{
  bool retransmision=false;
  set <Timer *>::iterator index;

//Comprobamos que realmente el segmento a retransmitir es uno
//almacenado en la lista de retransmisión, en cuyo caso damos luz verde
//al envío y eliminamos el segmento de la lista de retransmisión.
  if (!(timerTCP_.size()==0))
      {
        for ( index = timerTCP_.begin() ;index != timerTCP_.end() ;
index++)
```

```
            {
              if ((((*index)->packet())-
>PacketTCP::datos.seq_number)==p->PacketTCP::datos.seq_number)
                    {
                      (*index)->deactivate();
                      timerTCP_.erase(index);
                      retransmision=true;
                    }
            }
      }
  if (retransmision)
      {
        //Recalculamos la ventana de transmisión al ocurrir un
//vencimiento del timeout.
        connIns_->winCalculate(false,true);

        p->setOrigen(this);
        p->toDown();
        p->setConnId(connId_);
        p->PacketTCP::datos.wndChange=true;
        p->PacketTCP::datos.newWnd=connIns_->TxonWindow();
        send(now,p);
      }
}


////////////////////////////////////////////////////////////
//
//  Método: TimeOut
//
//  Función: Saca los segmentos asentidos de la cola de retransmisión.
//
////////////////////////////////////////////////////////////
void
SourceTCP::TimeOut(double now, Packet *p, bool liberacion)
{
  set <Timer *>::iterator index;

  //En fase de liberación de la conexión eliminamos todos los elementos
aún pendientes en la cola de retransmisión.
  if (liberacion)
      {
        if (timerTCP_.size()>0)
            {
              for ( index = timerTCP_.begin() ;index  !=
timerTCP_.end() ; index++)
                    {
                      (*index)->deactivate();
                      timerTCP_.erase(index);
                    }
            }
      }

  else if ( (p->PacketTCP::datos.ack) && (timerTCP_.size()>0))
//Eliminamos todos los elementos asentidos.
      {
        int ack=p->PacketTCP::datos.ack_number;
```

```
        for ( index = timerTCP_.begin() ;index != timerTCP_.end() ;
index++)
            {
              if (((((*index)->packet())->PacketTCP::datos.seq_number)+
                     (((*index)->packet())-
>PacketTCP::datos.data_size))<=ack)
                   {
                     (*index)->deactivate();
                     timerTCP_.erase(index);
                   }
            }
      }
}

////////////////////////////////////////////////////////////
//
// Método: isfull
//
// Función: Introducir los datos de usuario en el segmento de datos
//hasta que llegue al tamaño máximo y en ese caso enviar.
//
////////////////////////////////////////////////////////////
bool
SourceTCP::isfull(double now, Packet *p)
{
  bool full=false;
  int ssize=connIns_->segmentSize();
  int limite=(connIns_->wndLimit())+1;
  segsize_+=p->size();
//Enviamos sólo en el caso de que se haya llenado el segmento, hayamos
llegado al tamaño máximo de la ventana en transmisión o se nos haya
indicado la función psh desde niveles superiores.
  if ((segsize_>=ssize)||(segsize_>=limite)||(p->PacketTCP::datos.psh))
      {
        if (segsize_>=limite)
            {
              p->PacketTCP::datos.data_size=limite;
              segsize_=segsize_-limite;
              stopSend_=true;
              wakeUp((void (Module::*)(double, Packet
*))&Source::stopSource,now);
            }
        else if (segsize_>=ssize)
            {
              p->PacketTCP::datos.data_size=ssize;
              segsize_=segsize_-ssize;
            }
        else
            {
              p->PacketTCP::datos.data_size=segsize_;
              segsize_=0;
            }
        full=true;
      }
  return full;
}
```

```
//////////////////////////////////////////////////////////////
//
// Método: stop
//
// Función: Método stop sobrecargado de la clase source con el objetivo
// de simular una liberación de llamada.
//
//////////////////////////////////////////////////////////////
void
SourceTCP::stop (double tstop)
{
     wakeUp ((void (Module::*)(double, Packet *))
&SourceTCP::stopSourceTCP, tstop);
                  Source::stop(tstop);
}

//////////////////////////////////////////////////////////////
//
// Método: stopSourceTCP
//
// Función: Cuando la fuente comienza la liberación de llamada, este
// método envía un segmento con el bit fin activado.
//
//////////////////////////////////////////////////////////////
void
SourceTCP::stopSourceTCP (double tstop)
{
  TimeOut(tstop,0,true);
  printf ("\n CLOSE ( source )\n\n");
  Packet * p=connIns_->CLOSE();
  p->setOrigen(this);
  p->setConnId(connId_);
  send (tstop,p);
}

//////////////////////////////////////////////////////////////
//
// Método: stateClose
//
// Función: Poner el estado de la conexión a CLOSED una vez que la
//fuente estra en estado Time_Wait.
//
//////////////////////////////////////////////////////////////
void
SourceTCP::stateClose(double now, Packet * p)
{
  connIns_->putClosed();
}
```

8.2.3. SINK_TCP

SINK_TCP.H

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t
-*- */

#ifndef _sinkTCP_h_
#define _sinkTCP_h_

////////////// Includes del sistema
#include <set>
#include <list>
#include <iostream>


////////////// Includes de la aplicación
#include "node.h"
#include "module.h"
#include "distrib.h"
#include "connection.h"
#include "timer.h"

////////////// Defines del módulo
#define CONTADOR 1.5  //Marca un umbral para el contador de
retransmisiones.


class SinkTCP : public Node
{
public:
  SinkTCP  (double rbin, Queue * queueIns, int win_sink, unsigned int
numServ = 1, int segmentsize = 5000);
  virtual         ~SinkTCP ();

//Método encargado de procesar los segmentos de entrada y enviar en
//caso necesario al otro extremo de la conexión.
  virtual void    recvDown   (double now, Packet * p);

protected:

//Simula una llamada de usuario CLOSE cuando el nodo destino TCP se
encuentra en estado CLOSE_WAIT.
void    sndCLOSE (double tclose, Packet * p);

private:

//Método que establece el tamaño final del segmento a enviar al otro
//extremo en base al tamaño máximo de segmento para sink y a las
//posiciones libres de la ventana de transmisión. Devuelve true en caso
//de que haya posiciones libres en la ventana y podamos enviar el
//segmento.
  bool    isfull   (Packet * p);


//Método que reenvía un determinado segmento si ha expirado el contador
//de retransmisión.
  void    retxon   (double now, Packet * p);
```

```
//Cuando llega un asentimiento válido elimina los segmentos asentidos
//de la cola de retransmisión.
  void    TimeOut  (double now, Packet * p,bool liberacion);


//Instancia de la clase ConnectionTCP->se encarga de los parámetros de
//la conexión.
  Connection    * connIns_;

//Lista de timers de retransmisión. Se activa un timer y se incluye en
la lista cada vez que se produce un envío de datos.
  set <Timer *>       timerTCP_;

//controla el momento del envio para calcaular las posiciones de la
ventana de recepción.
  bool           firstsend_;
};

#endif
```

SINK_TCP.CC

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t
-*- */

#include "sinkTCP.h"
#include "scheduler.h"


SinkTCP::SinkTCP(double rbin, Queue *queueIns, int win_sink, unsigned
int numServ, int segmentsize) :
  /*Creamos una instancia de Node, ya que SinkTCP no es más que un tipo
especial de Nodo-> el destino de los segmentos de información*/
  Node(rbin,queueIns,numServ),
  firstsend_(true)
{

 /*Establecemos el primer número de secuencia del Nodo destino*/
  Distrib * issnum=new Exponential(3500);
  DistStat  issn  =issnum;
  connIns_=new Connection((int)issn.pickValue(), win_sink,
segmentsize);

  /*Simulamos una llamada de usuario OPEN pasiva, de modo que el Nodo
destino pase de un estado CLOSED ( que suponemos que tiene en un
principio ) al estado LISTEN, esperando información de la fuente*/
  connIns_->passiveOPEN();
}

SinkTCP::~SinkTCP()
{
}

////////////////////////////////////////////////////////////
//
```

```
// Método: recvDown
//
// Función: Es necesario sobrecargar el método recvDown de Node ya que
//necesitamos responder a la fuente, ya sea con el paquete de control
//necesario durante el establecimiento o la liberación de la llamada, o
//mandando datos .
//
////////////////////////////////////////////////////////////
void
SinkTCP::recvDown(double now, Packet *p)
{
    // En primer lugar hay que comprobar que el segmento recibido no es
un mensaje especial reset o no está dañado
  if (!(p->PacketTCP::datos.error_header))
   {
      if (p->PacketTCP::datos.rst)
            {
              connIns_->reset();

           //En el caso de que se haya recibido un reset durante un
//estado sincronizado habría que abortar la conexión. No podemos
//abortarla desde sink, así que el único modo de abortarla será
//indicárselo a la fuente de datos. Construimos un mensaje especial con
//el bit abort de cabecera a true.
              if (connIns_->abort())
                  {
                    Packet * abort=connIns_->errorMsg(0,true);
                    abort->toDown();
                    abort->setOrigen(this);
                    abort->setConnId(connId_);
                    Node::recvUp(now,abort);
                  }
            }
        else
            {

              //Antes de nada comprobamos si ha habido algún cambio en
//la ventana de recepción originado en el otro extremo a causa de un
//estado de congestión de la red. Debemos comprobarlo antes de procesar
//el segmento de entrada, puesto que la ventana me marca el límite para
//el número de secuencia del segmento entrante.
              if (p->PacketTCP::datos.wndChange)
                  connIns_->putWindow(p->PacketTCP::datos.newWnd);

              //PackProcessing es la máquina de estados de la conexión,
//su misión es comprobar el segmento de entrada y si éste no es
//inaceptable, actualizar el estado de la conexión y las variables
//asociadas a los espacios de números de secuencia y asentimiento, así
//como elaborar el segmento a enviar al otro extremo como respuesta al
//recibido.
              Packet * pack=connIns_->PackProcessing(p);
              if (!(pack==0))
                  {
                    if (connIns_->isconnEstab())
                        {
                          //  En este momento tenemos la llamada
//establecida, lo que hacemos es mandar a la fuente los mismos datos
```

```
//que hemos recibido con el objetivo de simular una comunicación full-
//duplex.
                        TimeOut(now,p,false);
                        bool newWnd=connIns_-
>winCalculate(firstsend_,false);
                        firstsend_=false;
                        if (isfull(pack))
                            {
                              if (newWnd)
                                  {
                              pack->PacketTCP::datos.wndChange=true;
                              pack->PacketTCP::datos.newWnd=connIns_-
>TxonWindow();
                                  }
                              connIns_->seqNumProcessing(pack);
                              pack-
>PacketTCP::datos.ack_number=connIns_->putAck();
                              pack->toDown();
                              pack->setOrigen(this);
                              Node::recvUp(now,pack);
                              Timer * t_aux=new
Timer(this,now+(double)CONTADOR,((void (Module::*)(double,Packet
*))&SinkTCP::retxon),pack);
                              (Scheduler::instance())->wakeUp(t_aux);
                              timerTCP_.insert(t_aux);
                            }
                       }
                  else
                       {
                         //Cuando sink tiene la conexión establecida y
//le llega una petición e liberación de la conexión, lo notifica al
//usuario de capas superiores, para que cuando haya terminado de enviar
//sus datos de usuario le solicite un CLOSE, mandando así un paquete
//fin al otro extremo. Simulamos todo esto mediante una llamada al
//método de sinkTCP sndCLOSE.   En este caso estamos en fase de
//establecimiento o liberación de llamada -> intercambiamos los bits de
//control necesarios que ya hemos puesto en el msg en el método
//checkPacket de la clase connection.
                        if (connIns_->isconnClWait())
                            {
                              TimeOut(now,0,true);
                              wakeUp((void (Module::*)(double, Packet
*))&SinkTCP::sndCLOSE,
                                          now+0.5);
                            }
                        //El envío del ack lo realizamos antes para
asentir el fin recibido antes de mandar el propio.
                        pack->setOrigen(this);
                        pack->setConnId(connId_);
                        pack->PacketTCP::datos.ack_number=connIns_-
>putAck();
                        Node::recvUp(now,pack);
                       }
                }
             else if (connIns_->returnError())
                {
                  Packet * error=connIns_->errorMsg(p,false);
```

```
                    error->setOrigen(this);
                    error->setConnId(connId_);
                    Node::recvUp(now,error);
                  }
              }
      }
}

////////////////////////////////////////////////////////////////
//
// Método: sndCLOSE
//
// Función: Simula una llamada de usuario CLOSE cuando el nodo destino
//TCP se encuentra en estado CLOSE_WAIT.
//
// Parámetros:
//   tclose -> instante en que se produce el envío de liberalización de
//la llamada.
//
////////////////////////////////////////////////////////////////
void
SinkTCP::sndCLOSE(double tclose, Packet *)
{
  printf ("\n CLOSE ( sink )\n\n");
  Packet * p=connIns_->CLOSE();
  p->setOrigen(this);
  p->setConnId(connId_);
  Node::recvUp(tclose,p);
}

////////////////////////////////////////////////////////////////
//
//  Método: isfull
//
//   Función: establece el tamaño del segmento a enviar por sink
//dependiendo del tamaño máximo de segmento para sink y el número de
//octetos libres en la ventana de transmisión. Devuelve TRUE si tenemos
//posiciones libres y podemos enviar el segmento.
//
////////////////////////////////////////////////////////////////
bool
SinkTCP::isfull(Packet *p)
{
  bool enviar=true;
  int limite=(connIns_->wndLimit())+1;
  int ssize=connIns_->segmentSize();
  if (limite>0)
      {
        if (ssize<limite)
            p->PacketTCP::datos.data_size=ssize;
        else
            p->PacketTCP::datos.data_size=limite;
      }
  else
      enviar=false;
  return enviar;
}
```

```
////////////////////////////////////////////////////////////
//
// Método: retxon
//
//  Función: Reenvía el segmento si ha expirado el contador de
//retransmisión.
//
////////////////////////////////////////////////////////////
void
SinkTCP::retxon(double now, Packet *p)
{
  bool retransmision=false;
  set <Timer *>::iterator index;

//Comprobamos que realmente el segmento a retransmitir es uno
//almacenado en la lista de retransmisión, en cuyo caso damos luz verde
//al envío y eliminamos el segmento de la lista de retransmisión.
  if (!(timerTCP_.size()==0))
      {
        for ( index = timerTCP_.begin() ;index != timerTCP_.end() ;
index++)
            {
              if ((((*index)->packet())-
>PacketTCP::datos.seq_number)==p->PacketTCP::datos.seq_number)
                  {
                    (*index)->deactivate();
                    timerTCP_.erase(index);
                    retransmision=true;
                  }
            }
      }

  if (retransmision)
      {
//Recalculamos la ventana de transmisión al ocurrir un vencimiento del
//timeout.
        connIns_->winCalculate(false,true);
        p->setOrigen(this);
        p->toDown();
        p->setConnId(connId_);
        p->PacketTCP::datos.wndChange=true;
        p->PacketTCP::datos.newWnd=connIns_->TxonWindow();
        Node::recvUp(now,p);
      }
}

////////////////////////////////////////////////////////////
//
//  Método: TimeOut
//
//  Función: Saca los segmentos asentidos de la cola de retransmisión.
//
////////////////////////////////////////////////////////////
void
SinkTCP::TimeOut(double now, Packet *p,bool liberacion)
{
```

```
  set <Timer *>::iterator index;

  //En fase de liberación de la conexión eliminamos todos los elementos
aún pendientes en la cola de retransmisión.
  if (liberacion)
      {
        if (timerTCP_.size()>0)
            {
              for ( index = timerTCP_.begin() ;index  !=
timerTCP_.end() ; index++)
                  {
                    (*index)->deactivate();
                    timerTCP_.erase(index);
                  }
            }
      }

//Eliminamos todos los elementos asentidos.
  else if ( (p->PacketTCP::datos.ack) && (!(timerTCP_.size()==0)))
      {
        int ack=p->PacketTCP::datos.ack_number;
        for ( index = timerTCP_.begin() ;index != timerTCP_.end() ;
index++)
            {
              if (((((*index)->packet())->PacketTCP::datos.seq_number)+
                     (((*index)->packet())-
>PacketTCP::datos.data_size))<=ack)
                  {
                    (*index)->deactivate();
                    timerTCP_.erase(index);
                  }
            }
      }
}
```

8.2.4. CONNECTION

CONNECTION.H

```
/* -*-  Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t -*-
*/

#ifndef _connection_h_
#define _connection_h_

#include <limits.h>

//////////Includes del sistema
typedef unsigned long int NUM_;

//////////Includes de la aplicación
#include "module.h"
#include "packet.h"

//////////Defines del módulo
```

```
class Connection
{
public:
  Connection       (int niss, int wnd, int seg_size);
  virtual         ~Connection       ();

//Simula una llamada de usuario Open Activa para source.
  Packet *        activeOPEN       ();

//Simula una llamada de usuario Open Pasiva para sink.
  void            passiveOPEN      ();

//Comprueba si la conexión se encuentra en estado ESTABLISHED.
  bool            isconnEstab    () { return connstate_==ESTABLISHED; }

//Comprueba si la conexión se encuentra en estado CLOSE_WAIT.
  bool            isconnClWait   () { return connstate_==CLOSE_WAIT; }

//Comprueba si la conexión se encuentra en estado TIME_WAIT.
  bool            isconnTmWait   () { return connstate_==TIME_WAIT; }

//máquina de estados.
  Packet *        PackProcessing   (Packet *);

// Pone los valores adecuados a los bits de control de la cabecera TCP.
  Packet *        ControlPacket    (bool syn, bool ack, bool fin);

//Establece y actualiza el número de secuencia a enviar en cada caso.
  void            seqNumProcessing (Packet *p);

//Método que se encarga de comprobar si el número de secuencia recibido
//es el esperado en recepción, y en caso afirmativo actualizar el
//siguiente esperado.
  bool            seqNumCheck      (Packet *p);

//Comprueba si se ha producido alguno de estos dos errores en la etapa
//de comprobación del segmento.
  bool            returnError    () { return (reset_||asentimiento_); }

//Devuelve el número de asentimiento del segmento.
  int             putAck           () { return rcvNxt_; }

//Devuelve true si hay que abortar la conexión.
  bool            abort            () { return abort_; }

//Procesa un reset en recepción, actuando en consecuencia al estado en
//que se encuentre la conexión.
  void            reset            ();

//Devuelve el tamaño del segmento en octetos.
  int             segmentSize      () { return ssize_; }

//Se ha producido un error en un estado de la conexión y construimos un
//mensaje en consecuencia: o bien un reset si se ha producido durante
```

```
//un estado no sincronizado o un asentimiento si se ha producido
//durante un estado sincronizado.
  Packet *        errorMsg        (Packet *p, bool abort);

//Calcula las posiciones de la ventana y modifica el valor tanto de la
//ventana (en caso de congestión) como del segmento de datos en caso
//necesario.
  bool            winCalculate    (bool firstsend, bool congestion);

//Devuelve el número de octetos que aún se pueden enviar hasta
//completar la ventana de transmisión.
  int             wndLimit        ();

//Devuelve el valor real de la ventana de transmisión.
  int             TxonWindow      () { return txonWnd_; }

//Actualiza el valor de la ventana de recepción con el valor que me
envía el otro extremo.
  void            putWindow       (int wnd) { rcvWnd_=(NUM_)wnd; }

//Actualiza el estado de conexión a FIN_WAIT_1 y los valores del
segmento a enviar.
  Packet *        CLOSE           ();

//EStablece estado de conexión CLOSED.
  void            putClosed       ();


private:

// Posibles estados de una conexión:
  enum
      {
        CLOSED,
        LISTEN,
        SYN_SENT,
        SYN_RCVD,
        ESTABLISHED,
        FIN_WAIT_1,
        FIN_WAIT_2,
        TIME_WAIT,
        CLOSE_WAIT,
        LAST_ACK,
      } connstate_;

//Variables relativas al espacio de secuencia del transmisor:

//almacena el tamaño en octetos de la ventana en transmisión.
  NUM_      sndWnd_ ;
//siguiente nº de secuencia que se va a enviar.
  NUM_      sndNxt_;
//último número de secuencia asentido del transmisor
  NUM_      sndUna_;

//Variables relativas al espacio de secuencia del receptor:
```

```
//almacena el tamaño en octetos de la ventana en recepción.
  NUM_      rcvWnd_ ;
//siguiente nº de secuencia que espero en recepción.
  NUM_      rcvNxt_;

//Tamaño del segmento, en caso normal coincide con el siguiente
//parámetro, pero puede variar como consecuencia del tamaño de la
//ventana en recepción.
  NUM_      ssize_;
//Tamaño del segmento, dato pasado por el usuario por la línea de
//comandos o tomada por defecto.
  NUM_      ssizeInicial_;
//Tamaño de la ventana de recepción en cada instante, dependiendo del
//estado de congestión.
  NUM_      txonWnd_;
//Flag que indica cuando nos encontramos en un estado de recuperación
//después de un estado de congestión en la red.
  bool      recuperacion_;
//Flag que indica al transmisor si se ha generado un reset como
//resultado de la comprobación del segmento entrante.
  bool      reset_;
//Variable a true en caso de que haya que abortar la conexión por algún
//motivo.
  bool      abort_;
//Variable a true si ha llegado un segmento no acorde con el estado de
la conexión encontrándose éste en un estado sincronizado. Avisa al
extremo en cuestión que debe mandar un asentimiento con los datos
válidos del segmento esperado en recepción.
  bool      asentimiento_;
};

#endif
```

CONNECTION.CC

```
/* -*-      Mode:C++; c-basic-offset:2; tab-width:4; indent-tabs-mode:t
-*- */

#include "connection.h"

Connection::Connection(int niss, int wnd, int seg_size) :
  connstate_(CLOSED),
  recuperacion_(false),
  reset_(false),
  abort_(false),
  asentimiento_(false)
{
  sndNxt_= (NUM_) niss;
//ventana permitida para los segmentos en recepción.
  rcvWnd_= (NUM_) wnd;
  ssize_= (NUM_) seg_size;
  ssizeInicial_= (NUM_) seg_size;
}
```

```
Connection::~Connection()
{
}

//////////////////////////////////////////////////////////////
//
// Método: passiveOPEN
//
// Función: Cuando se produce una llamada de usuario OPEN pasiva, la
// conexión pasa a estado LISTEN.
//
//////////////////////////////////////////////////////////////
void
Connection::passiveOPEN()
{
  printf ("\nOPEN PASIVO ( sink ) \n");
  connstate_=LISTEN;
}

//////////////////////////////////////////////////////////////
//
// Método: activeOPEN
//
// Función: Cuando se produce una llamada de usuario OPEN activa, la
//conexión pasa a estado SYN_SENT y se envía una petición de conexión
//al otro extremo.
//
//////////////////////////////////////////////////////////////
Packet *
Connection::activeOPEN()
{
  printf ("\nOPEN ACTIVO ( source ) \n\n");
  connstate_=SYN_SENT;
  Packet * p=ControlPacket(true,false,false);
  seqNumProcessing(p);

//Envío al otro extremo el rango de números de secuencia que source
está dispuesto a aceptar.
  p->PacketTCP::datos.window=rcvWnd_;
  return (p);
}

//////////////////////////////////////////////////////////////
//
// Método: CLOSE
//
// Función: Actualizar le estado de la conexión cuando se comienza a
//liberar la llamada, así como el segmento de control a enviar.
//
//////////////////////////////////////////////////////////////
Packet *
Connection::CLOSE ()
{
  if (connstate_==ESTABLISHED)
      {
        connstate_=FIN_WAIT_1;
        printf(" SOURCE : ESTABLISHED---->FIN_WAIT_1\n");
```

```
      }else
        {
            connstate_=LAST_ACK;
            printf(" SINK : CLOSE_WAIT---->LAST_ACK\n");
        }
  Packet * p=ControlPacket(false,true,true);
  p->PacketTCP::datos.psh=true;
  seqNumProcessing(p);
  p->PacketTCP::datos.ack_number=rcvNxt_;
  return (p);
}

////////////////////////////////////////////////////////////
//
// Método: ControlPacket
//
//  Función: En el proceso de establecimiento y liberalización de
//llamada hay que enviar varios paquetes que son simplemente de
//control. Este método tiene como objetivo introducir los datos de
//control en el paquete sean cuales sean éstos y así evitamos mucha
//repetición de código.
//
// Parametros:
//    syn : TRUE para activar el bit de cabecera syn.
//    ack : TRUE para activar el bit de cabecera ack.
//    fin : TRUE para activar el bit de cabecera fin.
//
// Devuelve:
//    El paquete a enviar.
//
////////////////////////////////////////////////////////////
Packet *
Connection::ControlPacket(bool syn, bool ack, bool fin)
{
  Packet * controlPaq = new Packet (0);
  controlPaq->PacketTCP::datos.data_size=0;
  controlPaq->PacketTCP::datos.error_header=false;
  controlPaq->PacketTCP::datos.abort=false;
  controlPaq->PacketTCP::datos.psh=false;
  controlPaq->PacketTCP::datos.rst=false;
  controlPaq->PacketTCP::datos.wndChange=false;
  controlPaq->toDown();
  controlPaq->PacketTCP::datos.syn=syn;
  controlPaq->PacketTCP::datos.ack=ack;
  controlPaq->PacketTCP::datos.fin=fin;
  return controlPaq;
}

////////////////////////////////////////////////////////////
//
// Método: seqNumProcessing
//
// Función: Método que se encarga de establecer el número de secuencia
//adecuado a cada paquete que se va a enviar, así como de actualizar
//los campos relativos a dichos números de secuencia.
//
////////////////////////////////////////////////////////////
```

```
void
Connection::seqNumProcessing (Packet * p)
{

  //Añadimos al segmento el número de secuencia a enviar, almacenado en
la variable sndNxt_.
  p->PacketTCP::datos.seq_number=sndNxt_;

  //Si el tamaño del segmento es distinto de cero se trata de un
//segmento de datos, en cuyo caso actualizamos el siguiente número de
//secuencia a enviar sumándole el tamaño del segmento.
  if (!(p->PacketTCP::datos.data_size==0))
      {
        sndNxt_+=p->PacketTCP::datos.data_size;
      }

  //En caso de que el tamaño del segmento sea cero tenemos dos
//posibilidades: que sea un asentimiento o un segmento de control
//intercambiado en el establecimiento o liberación de la conexión. En
//el primero de los casos el siguiente número de segmento a enviar no
//varía, por lo que no lo tenemos en cuenta. Si se trata del segundo
//caso, la variable sndNxt_ se incrementa en una unidad.
  else if ((p->PacketTCP::datos.syn)||(p->PacketTCP::datos.fin))
      {
        sndNxt_+=1;
      }
}

//////////////////////////////////////////////////////////////
//
//  Método: PackProcessing
//
//   Función: máquina de estados. Procesa el segmento de entrada:
//comprueba que no se trate de un segmento inaceptable según el estado
//de conexión en que nos encontremos y actua en consecuencia,
//actualizando el estado de la conexión y creando el segmento a enviar
//al otro extremo en respuesta al recibido.
//
//////////////////////////////////////////////////////////////
Packet *
Connection::PackProcessing(Packet * p)
{

//Comprueba si ha llegado un segmento inaceptable durante un estado no
sincronizado de la conexión.
  bool NO_SYN=true;
//Comprueba si ha llegado un segmento inaceptable durante un estado
sincronizado de la conexión.
  bool SYN=true;

  Packet * enviar=0;


  switch ( connstate_ )
      {
```

```
        // Estando en estado LISTEN sólo vamos a aceptar peticiones de
//conexión (ie, segmentos con el bit syn activo)-> si esto ocurre no
//hay que comprobar nada más.

      case LISTEN:
        {
            if (p->PacketTCP::datos.syn && !p->PacketTCP::datos.ack &&
!p->PacketTCP::datos.fin)
              {
                  connstate_=SYN_RCVD;
                  printf (" SINK : LISTEN---->SYN_RCVD\n");

                  //Preparamos el segmento a enviar como respuesta al
//recibido:
                  enviar=ControlPacket(true,true,false);
                  enviar->PacketTCP::datos.window=rcvWnd_;
                  seqNumProcessing(enviar);

                  //Modificamos    las    variables    necesarias    como
//consecuencia del segmento de entrada:
                  rcvNxt_=p->PacketTCP::datos.seq_number+1;
                  sndWnd_=p->PacketTCP::datos.window;
                  txonWnd_=sndWnd_;
              }
            else
              NO_SYN=false;
            break;
        }

        // En estado SYN_SENT y según la implementación del simulador
//en la que no está contemplada la iniciación de conexión simultánea,
//debería llegarnos del otro extremo un paquete SYN, con el
//asentimiento al SYN que se envió en el OPEN ACTIVO y el número de
//secuencia inicial. Habrá que verificar tanto que el segmento tenga el
//bit SYN activo, como que el número de ack sea el esperado.

      case SYN_SENT:
        {
            if ((p->PacketTCP::datos.syn && p->PacketTCP::datos.ack &&
!p->PacketTCP::datos.fin) && ( p-
>PacketTCP::datos.ack_number==sndNxt_))
              {
                  connstate_=ESTABLISHED;
                  printf (" SOURCE : SYN_SENT---->ESTABLISHED\n");
                  //Preparamos el segmento a enviar como respuesta al
//recibido:
                  enviar=ControlPacket(false,true,false);
                  seqNumProcessing(enviar);

                  //Modificamos    las    variables    necesarias    como
//consecuencia del segmento de entrada:
                  rcvNxt_=p->PacketTCP::datos.seq_number+1;
                  sndWnd_=p->PacketTCP::datos.window;
                  txonWnd_=sndWnd_;
                  sndUna_=p->PacketTCP::datos.ack_number-1;
              }
            else
```

```
              NO_SYN=false;
            break;
        }

        // Estando en estado SYN_RCVD sólo se aceptarán paquetes de
//asentimiento. Habrá que comprobar que tanto el nº de ack, como el nº
//de secuencia son los esperados.

      case SYN_RCVD:
        {
            if ((!p->PacketTCP::datos.syn && p->PacketTCP::datos.ack &&
!p->PacketTCP::datos.fin) && (p->PacketTCP::datos.seq_number==rcvNxt_)
&& (p->PacketTCP::datos.ack_number==sndNxt_))
              {
                  connstate_=ESTABLISHED;
                  printf (" SINK : SYN_RCVD---->ESTABLISHED\n");

                  //No hay ningún segmento que enviar en respuesta a
//este.

                  //Modificamos las variables necesarias como
//consecuencia del segmento de entrada:
                  sndUna_=p->PacketTCP::datos.ack_number-1;
              }
            else
              NO_SYN=false;
            break;
        }

        // En el caso del estado ESTABLISED, primer estado
//sincronizado, lo único inaceptable sería un SYN ( implicaría que ha
//llegado un segmento de una conexión antigua->imposible en el
//simulador, o que ha habido error ). En cualquier caso, se informa al
//usuario y se envía al otro extremo un asentimiento con los datos
//actuales de número de secuencia y asentimiento. Habrá que comprobar
//nº de secuencia y nº de ack.

      case ESTABLISHED:
        {

            if ((!p->PacketTCP::datos.syn) && (sndUna_<p-
>PacketTCP::datos.ack_number)&&(p-
>PacketTCP::datos.ack_number<=sndNxt_))
              {
                  if (seqNumCheck(p))
                    {
                        sndUna_=p->PacketTCP::datos.ack_number-1;
                        enviar=ControlPacket(false,true,false);
                        if (p->PacketTCP::datos.fin)
                          {
                              connstate_=CLOSE_WAIT;
                              printf(" SINK : ESTABLISHED----
>CLOSE_WAIT\n");
                              sndNxt_=p->PacketTCP::datos.ack_number;
                              rcvNxt_=p->PacketTCP::datos.seq_number+1;
                              seqNumProcessing(enviar);
                          }
```

```
                    }
                  else
                    asentimiento_=true;
                }
              else
                SYN=false;
              break;
          }

          //Si nos econtramos en estado FIN_WAIT_1 o LAST_ACK sólo vamos
//a aceptar segmentos con el bit ack activo. En el primer caso el nodo
//ha enviado un fin y está esperando que se le asienta; y en el segundo
//caso se ha recibido y asentido un segmento fin entrante, pero no
//cerramos la conexión hasta haber enviado todos los segmentos de
//datos, por tanto debemos aceptar los asentimientos de todos los
//paquetes enviados en este tiempo. Por supuesto hay que verificar en
//ambos casos que tanto los números de secuencia, como los de
//asentimiento son los esperados.

        case FIN_WAIT_1:
        case LAST_ACK:
          {

              if ((!p->PacketTCP::datos.syn && p->PacketTCP::datos.ack &&
!p->PacketTCP::datos.fin) && (p->PacketTCP::datos.ack_number==sndNxt_))
                {
                    if  (p->PacketTCP::datos.seq_number==rcvNxt_)
                      {
                          if (connstate_==FIN_WAIT_1)
                            {
                                printf (" SOURCE : FIN_WAIT_1----
>FIN_WAIT_2\n");
                                connstate_=FIN_WAIT_2;
                            }
                          else
                            {
                                printf (" SINK : LAST_ACK---->CLOSED\n");
                                connstate_=CLOSED;
                            }

                          //No  se  envía  nada  en  respuesta  al  segmento
//recibido.

                          //Se atualizan las variables necesarias.
                          rcvNxt_=p->PacketTCP::datos.seq_number+1;
                          sndUna_=p->PacketTCP::datos.ack_number-1;
                      }
                    else
                          asentimiento_=true;
                }
              else
                SYN=false;
          break;
        }

          //En  el  caso  del  estado  FIN_WAIT_2,  serían  aceptables  tanto
//segmentos con bit fin activo como con el bit ack activos, ya que en
```

```
//este estado nos encontramos cuando hemos mandado el bit fin propio y
//estamos esperando que el otro extremos termine de enviarnos sus datos
//para que nos envíe el fin.

      case FIN_WAIT_2:
        {

            if (!p->PacketTCP::datos.syn  && (p-
>PacketTCP::datos.ack_number==sndNxt_))
              {
                  connstate_=TIME_WAIT;
                  printf(" SOURCE : FIN_WAIT_2---->TIME-WAIT\n");

                  //Preparamos el segmento a enviar como respuesta al
//recibido:
                  enviar=ControlPacket(false,true,false);
                  seqNumProcessing(enviar);

                  //Modificamos    las    variables    necesarias    como
//consecuencia del segmento de entrada:
                  rcvNxt_=p->PacketTCP::datos.seq_number+1;
                  sndUna_=p->PacketTCP::datos.ack_number-1;
              }
            else
                  SYN=false;
            break;
        }

      case CLOSE_WAIT:
      case TIME_WAIT:
        break;
      }
  if (!NO_SYN)
      {
        printf ("\n\t **ERROR**\n Fallo durante el establecimiento de
llamada-> Ha llegado un segmento no acorde con el estado de la
conexión. Se enviará un RESET al otro extremo con lo que finalizará la
simulación.\n\n");
        reset_=true;
      }

  //Si se produce un error debido a un segmento no esperado durante un
//estado sincronizado hay que enviar al otro extremo un segmento de
//asentimiento con los datos esperados de número de secuencia y de
//asentimiento correspondientes.

  if (!SYN)
      {
        printf ("\n\t **WARNING**\n La conexión se encuentra en un
estado sincronizado y ha llegado un segmento de datos no acorde con
dicho estado.\n\n");
      }
  return (enviar);
}
```

```
//////////////////////////////////////////////////////////////
//
// Método: putClosed
//
// Función: Establecer estado de la conexión CLOSED en caso necesario.
//
//////////////////////////////////////////////////////////////
void
Connection::putClosed()
{
  printf(" SOURCE : TIME_WAIT---->CLOSED\n");
  connstate_=CLOSED;
}

//////////////////////////////////////////////////////////////
//
// Método: seqNumCheck
//
//  Función:  Método  que  se  encarga  de  comprobar  si  el  número  de
//secuencia recibido es el esperado en recepción, y en caso afirmativo
//actualizar el siguiente esperado.
//
//////////////////////////////////////////////////////////////
bool
Connection::seqNumCheck(Packet *p)
{
  bool ok=false;
  int size=p->PacketTCP::datos.data_size;
  int seqNum=p->PacketTCP::datos.seq_number;

  if (size==0)
      {
        if ((rcvWnd_==0)&&(seqNum==rcvNxt_))
              ok=true;
        if ((rcvWnd_>0)&&(rcvNxt_<=seqNum)&&(seqNum<(rcvNxt_+rcvWnd_)))
            ok=true;
      }

  else if (rcvWnd_>0)
      {
        if
((rcvNxt_<=seqNum)&&(seqNum<(rcvNxt_+rcvWnd_))||(rcvNxt_<=(seqNum+size-
1))&&((seqNum+size-1)<(rcvNxt_+rcvWnd_)))
            ok=true;
      }
  if (ok)
      rcvNxt_=seqNum+size;
  return (ok);
}
```

```
////////////////////////////////////////////////////////////
//
//  Método: reset
//
//   Función: procesar un mensaje especial reset en recepción, actuando
//consecuentemente al estado en que se encuentre la conexión.
//
////////////////////////////////////////////////////////////
void
Connection::reset()
{
  if ((connstate_ ==SYN_SENT)||(connstate_==SYN_RCVD))
      connstate_=LISTEN;

//Si no es ninguno de los casos anteriores es que la conexión se
//encuentra en un estado sincronizado.
  else {
      printf("\n\t **ERROR**\n Ha llegado un mensaje RESET a uno de los
extremos estando la conexión en un estado sincronizado, por lo que se
va a proceder a abortar la conexión.\n\n");
      abort_=true;
  }
}

////////////////////////////////////////////////////////////
//
//  Método: errorMsg
//
//   Función: Se ha producido un error en un estado de la conexión y
//construimos un mensaje en consecuencia: o bien un reset si se ha
//producido durante un estado no sincronizado o un asentimiento si se
//ha producido durante un estado sincronizado.
//
////////////////////////////////////////////////////////////
Packet *
Connection::errorMsg(Packet * p, bool abort)
{
  Packet * error=0;

  if (reset_)
      {
        error=ControlPacket(false,false,false);
        error->PacketTCP::datos.rst=true;
        error->PacketTCP::datos.seq_number=p-
>PacketTCP::datos.ack_number;
        reset_=false;
      }

  if (asentimiento_)
      {
        error=ControlPacket(false,true,false);
        error->PacketTCP::datos.seq_number=p-
>PacketTCP::datos.ack_number;
        error->PacketTCP::datos.ack_number=rcvNxt_;
        sndNxt_=p->PacketTCP::datos.ack_number;
        if (p->PacketTCP::datos.wndChange)
            rcvWnd_=p->PacketTCP::datos.newWnd;
```

```
        asentimiento_=false;
      }

  if (abort)
      {
        error=ControlPacket(false,false,false);
        error->PacketTCP::datos.abort=true;
      }
  return (error);
}

////////////////////////////////////////////////////////////
//
//  Método: winCalculate
//
//   Función: Establecer el tamaño de la ventana según el tráfico de
//datos recibidos y, en caso necesario, modificar el tamaño del
//segmento.
//
//  Parámetros:
//    firstsend -> true en caso de que sea la primera vez que entramos
//en estado established. En este caso hay que dar valor inicial a todas
//aquellas variables relacionadas con la gestión de la ventana.
//       congestion -> true en caso de que se haya producido la
//retransimión de un segmento como consecuencia del vencimiento del
//contador de retransmisión.
//
////////////////////////////////////////////////////////////
bool
Connection::winCalculate(bool firstsend, bool congestion)
{
  bool wndChange=false;

  if (firstsend)
      {
        //Se puede dar la opción, al menos en Sink, de que el usuario
//establezca una ventana en recepción cero, o un tamaño de segmento
//cero, en el caso de que no quiera que el destine mande datos de
//usuario al otro extremo.

        //ERROR-->Lo notificamos al usuario y fijamos longitud de
//segmento cero para poder continuar la simulación.

        if ((sndWnd_==0) && (ssize_>0))
            {
              printf("\n\t ***ERROR*** Es inaceptable una ventana en
transmisión nula y una longitud de segmento mayor que cero. Para seguir
la simulación se fijará el tamaño del segmento a cero octetos, con lo
que sólo se transmitirán segmentos de asentimiento.\n\n");
              ssize_=0;

//Estado normal, ventana en recepción y tamaño de segmento mayor que
//cero.
            }else
              if (ssize_>sndWnd_)
                  ssize_=sndWnd_;
```

```
      //Entramos aquí en caso de que haya ocurrido una retransmisión
//como consecuencia del vencimiento de un TimeOut->deducimos que hay
//congestion.

      } else if ( congestion) {
        recuperacion_=true;

        //Dividimos la ventana a la mitad, alejándonos rápidamente de
//situaciones de saturación.

        txonWnd_=txonWnd_/2;
        if (txonWnd_ == 0)
            ssize_=0;
        else
            if ( ssize_ > txonWnd_ )
              ssize_=txonWnd_;

// Entramos aquí en caso de que estemos en un proceso de recuperación
//de ventana después de un estado de congestion.

      } else if ( recuperacion_ ) {

//Nº de octetos que voy a incrementar en la ventana.
        int incremento=(int)((ssize_*ssize_)/txonWnd_);

//Actualizamos la ventana con el incremento.
        txonWnd_+=incremento;

//Tamaño del segmento en caso de no congestion.
        int sizeaux=(ssizeInicial_>sndWnd_) ? sndWnd_ : ssizeInicial_;
        if (txonWnd_>=sndWnd_)
            {
              txonWnd_=sndWnd_;
              recuperacion_=false;
              ssize_=sizeaux;
            } else
              if ((ssize_+=incremento)>sizeaux)
                  ssize_=sizeaux;
        wndChange=true;
      }
  return (wndChange);
}

////////////////////////////////////////////////////////////
//
//  Método: wndLimit
//
//   Función : devolver como resultado el número de octetos que aún
//pueden ser enviados hasta completar la ventana de transmisión.
//
////////////////////////////////////////////////////////////
int
Connection::wndLimit()
{
      return ((sndUna_+txonWnd_)-sndNxt_);
}
```