CAPÍTULO 5 : DESARROLLO DEL SIMULADOR

Este proyecto expone el trabajo desarrollado para implementar un simulador del estándar TCP, comentado brevemente en el segundo capítulo de esta memoria, basado en el lenguaje de programación orientado a objetos C++, cuyas características principales se resaltan en el tercer capítulo. Puesto que se trata de un simulador y no de una implementación real, habrá muchos conceptos que queden fuera del ámbito del simulador. Todas las suposiciones bajo las que se basa el simulador se irán detallando a medida que se analice el código.

Como se comentó en la introducción de este documento, el simulador objeto del proyecto no es un código aislado, sino que es una parte independiente dentro de un código más amplio. Como se ha desarrollado en el tercer capítulo, dos de las grandes ventajas de la OOP son la herencia y el polimorfismo, que nos van a permitir utilizar las características del código del simulador de datos básico que se nos pasa como punto de comienzo de nuestro simulador.

5.1. SIMULADOR BÁSICO

En este punto se comentarán aquellas características sobre las que se basa el funcionamiento del programa que se desarrolla a continuación, así como los cambios necesarios sobre este código con el objetivo de caracterizar por completo el estándar TCP. El código completo de cada uno de los métodos analizados a continuación se encuentra en el primer anexo de esta memoria.

Incluso para una conexión tan sencilla como la especificada en la figura 5, intervienen gran número de clases para lograr el intercambio de información entre los distintos nodos. Para modelar este esquema es necesario identificar tres grupos fundamentales: en un primer grupo habría que englobar los distintos nodos que forman parte del esquema de conexión (independientemente de cómo sea éste), hay que dar formato a la información que se van a intercambiar y por último hay que modelar el intercambio de dicha información. Vamos viendo las clases que intervienen en cada grupo.

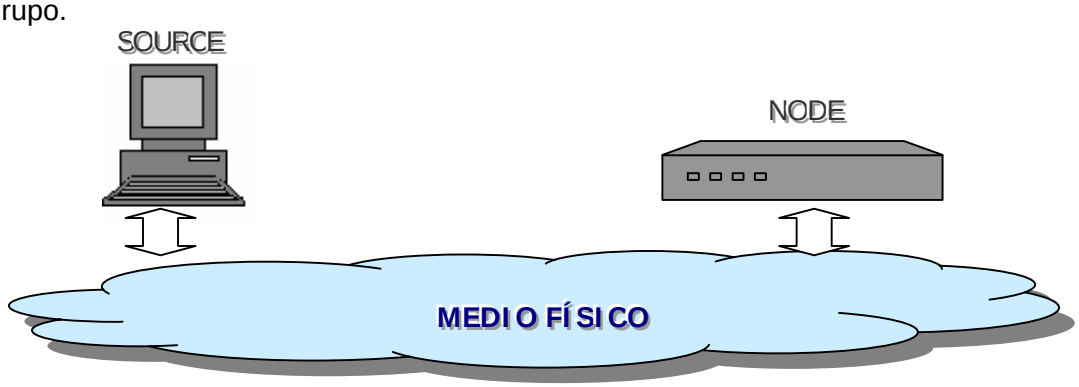

Figura 5. Conexión simple entre terminales

5.1.1. NODOS EN EL ESQUEMA DE CONEXIÓN

En el esquema más básico ilustrado en la figura 5 nos encontramos tres tipos de nodos: la fuente de la información, los nodos intermedios por los que se encaminan los datos y el medio físico.

◆ La fuente que emite los datos se define en la clase `source`. Tiene dos misiones fundamentales:

- Controlar el comienzo y finalización del envío de datos. Se usa para ello cuatro métodos: `start`, `startSource`, `stop` y `stopSource`. El instante de comienzo de envío de datos coincide con el comienzo de la simulación y es fijado por el programador en el main, mientras que la finalización del envío de datos coincide con el final de la simulación y es el usuario del programa quien lo define en la línea de comandos.

- Generar los paquetes de datos cada vez que transcurre un tiempo entre llegadas y enviarlos al otro extremo. Se define un sólo método para llevar a cabo este cometido, el método `nextArrival`. Se trata de un método recursivo, cuya función es enviar datos siempre que estemos dentro del tiempo de simulación y volver a llamarse transcurrido un tiempo entre llegadas de paquetes.

La fuente de datos TCP heredará, como es lógico pensar, las características de source. El código original de esta clase ha sufrido una ligera modificación durante el desarrollo del simulador, concretamente se ha añadido dos nuevos métodos: **fillPacket** y **putPsh**.

En el capítulo dos, punto 2.6 del presente documento (comunicación de datos) se mencionó el hecho de que el nivel TCP no envía los datos al otro extremo tal y como le llegan de las capas superiores, sino que los almacena a su conveniencia en segmentos cuya longitud no se especifica en ningún momento en el estándar. Puesto que nextArrival envía directamente los datos de usuario, resulta necesario “frenar” de algún modo el envío hasta que el segmento no esté completamente lleno. Con este fín se declara el nuevo método fillPacket que devuelve true si el segmento está lleno y se puede enviar, por lo que antes de cada envío se comprueba el resultado de su llamada.

Para no modificar el comportamiento por defecto de la fuente, se fuerza en source a que el método devuelva siempre true y se sobrescribe en la fuente de datos TCP modificando a nuestra conveniencia su modo de actuar.

El usuario transmisor de los datos, para asegurar que los datos que está enviando a través de TCP están llegando a su destino, define una función push que hace que el TCP envíe los datos que tiene al TCP destino independientemente de si se ha llegado o no a completar un segmento. Cuando el nivel superior indica una función push, el TCP envía inmediatamente los datos al TCP del otro extremo, y le indica activando el flag de cabecera psh que también él debe entregar sin demora los datos al usuario. Mediante el nuevo método putPsh modelamos este comportamiento: cada vez que se crea un nuevo paquete en nextArrival se llama a este método cuyo único

objetivo es poner a true el bit psh de cabecera de los datos. En el momento en que el usuario de nivel superior quisiera indicar una función psh no tendría más que sobrescribir este método y cambiar el valor del bit.

- Los distintos nodos intermedios por los que atraviesa el paquete de datos hasta que llega a su destino se definen mediante la clase `node`. En realidad el comportamiento de los nodos resulta transparente a la implementación del simulador, a no ser por el hecho de que el sumidero donde van los datos, el destino de la comunicación, no es otra cosa que un tipo especial de nodo. Por lo que nuestra clase destino de datos TCP heredará node.

Hay que destacar de esta clase el método `recvUp`, que se encarga de procesar el paquete de datos de entrada comprobando antes de su posterior envío que haya algún servidor libre. En el caso de que no hubiera ningún servidor libre se incluye el paquete en la cola de retransmisión si hay espacio disponible, si no se perderían los datos. La clase encargada de la gestión de la cola de paquetes es `queue`. Los métodos que componen esta clase son fundamentalmente `push` (mete el paquete de datos en la cola siempre que ésta no esté llena), `pop` (saca un paquete de la cola) y size (devuelve el número de paquetes en la cola).

Además del método recvUp, que es el más interesante para el simulador, node también implementa los métodos `nextDeparture`, que se encarga del envio del paquete de datos, y `getstatistics` que devuelve los estadísticos de los paquetes recibidos y perdidos.

- Por último el medio físico viene caracterizado mediante la clase `media`. Su modo de operación resulta transparente para el simulador, aún así comentar que su función es simplemente cambiar la dirección del paquete de datos al reenviarlo.

5.1.2. PAQUETE DE DATOS

Los paquetes de datos vienen definidos en la clase `packet`, que permite operar sobre ellos. Para definir el contenido de los paquetes se implementan tres clases a partir de Packet: `PacketIP`, `PacketTCP` y `ATMCell` donde cada protocolo puede definir sus datos.

Las operaciones permitidas sobre los paquetes de datos desde packet son obtener la dirección del paquete (`isUp`) y modificarlo (`toUp`, `toDown`), conocer el módulo origen de los datos (`origen`) y modificarlo (`setOrigen`), obtener el tamaño de los datos (`size`) y modificarlo (`setSize`), obtener el identificador de conexión sobre la que van los datos (`connId`) y modificarlo (`setConnId`), y por último obtener los datos (`data`), aunque por tratarse de un simulador, no se saben los datos en sí, sino solamente su tamaño.

Para la implementación de nuestro simulador se definen los datos en la estructura data de la clase PacketTCP, añadiendo la información necesaria de cabecera para procesar los segmentos TCP.

Entre los datos que componen el segmento TCP se encuentran:

- **error_header**, una variable tipo booleano que comprueba si hay algún tipo de error en la cabecera al comprobar el checksum.
- **data_size**, variable entera que almacena el tamaño de los datos.
- **seq_number**, entero que almacena el número de secuencia del primer octeto de datos, esto es, el número de serie del segmento.
- **ack_number**, entero que almacena el número de asentimiento del segmento, ie. el siguiente número de secuencia esperado en recepción.
- **window**, variable tipo entero que almacena el número de octetos de datos empezando por el indicado en el campo ack_number que el transmisor del segmento está dispuesto a aceptar.
- **newWnd**, variable significativa si está activo el bit wndChange de cabecera, indica el nuevo valor de la ventana de recepción para el TCP receptor del segmento. El TCP que detecta un estado de congestión en la red actúa disminuyendo su ventana de transmisión con el fin de delimitar el envío y salir de este estado. Al modificar su ventana de transmisión está modificando la ventana de recepción del TCP del otro extremo, puesto que se trata de la misma ventana. Mediante esta variable el TCP que detecta la congestión avisa al otro extremo que ha modificado su ventana y le indica el nuevo valor de la misma.

Y por último una serie de flags que son:

- **abort**, que toma valor positivo en caso de que haya que abortar la conexión.
- **ack**, positivo si el campo ack_number es significativo.
- **psh**, positivo si se señala la función push.
- **rst**, toma valor positivo si se envía un mensaje especial reset al otro extremo.
- **syn**, positivo si se están sincronizando los números de secuencia durante el establecimiento de la conexión.
- **fin**, que toma valor positivo para informar al otro extremo que no hay más datos que enviar.
- **wndChange,** bit positivo si se ha modificado la ventana de transmisión del TCP emisor del segmento, esto es, indica al TCP receptor que ha cambiado el valor de su ventana de recepción.

5.1.3. INTERCAMBIO DE INFORMACIÓN

Las clases implicadas en el envío de datos entre nodos son `schedule`, `module` y `timer`. Su comprensión resulta dura aunque de vital importancia para poder desarrollar nuestro código.

Scheduler representa la espina dorsal del simulador. Desde ella da comienzo la simulación y lleva toda la carga de la misma. Su método fundamental es `run`. Scheduler define una cola de elementos tipo Timer en la que se van almacenando los distintos eventos que van a ocurrir, ordenados según el instante de ejecución, como el envío de un paquete. Run va sacando de forma ordenada estos eventos y los va ejecutando.

Timer es la clase genérica de temporizadores. Resulta conveniente conocer la información que almacena cada elemento timer y que se le pasa al constructor puesto que es ésta la información que maneja scheduler para ejecutar un evento. La clase define dos posibilidades con respecto al constructor, uno al que no hay que pasar parámetros: los asigna todos a cero por defecto y no activa ningún temporizador; y otro que pone la variable active_ a true (activa de este modo el temporizador), y al que se le pasa como parámetro el Module (nodo o fuente de datos) origen del evento, el método a ejecutar por dicho Module, el paquete de datos implicado en el método y el momento del inicio del evento.

La clase timer define dos métodos fundamentales a la hora de utilizarla como temporizador: `timeOut`, desactiva el temporizador y ejecuta el método almacenado en la instancia de la clase si el temporizador estaba activo; y `deactivate`, que desactiva el temporizador y elimina la información del paquete asociado al objeto timer.

Durante el desarrollo del nuevo código también se ha tenido que añadir un nuevo método a esta clase: **packet**. Para implementar el contador de retransmisión se hace uso de la clase timer como contador, como se verá en siguientes capítulos. Se define una lista de timers donde se van a ir almacenando los segmentos enviados y no se eliminarán de la lista hasta que no se reciba asentimiento o se reenvíe el segmento porque ha expirado el contador. Cada vez que llega un asentimiento a uno de los extremos de la conexión, se deben hacer comparaciones con el número de asentimiento del segmento entrante y con características de los segmentos almacenados en la lista de timers para ver cuales están asentidos y eliminarlos de la lista, sin embargo la clase timer no ofrece ninguna posibilidad de tener acceso al paquete de datos almacenado en él. De aquí surge la necesidad de definir el nuevo método packet, que devuelve el valor de la variable de instancia packet_, que almacena el segmento de datos asociado al elemento timer.

Por último hay que destacar el método Module. La completa comprensión de esta clase se antoja esencial, ya que tanto Node como Source son implementaciones de Module y, como tal, utilizan cada uno de sus métodos.

Module se encarga básicamente de la relación de los nodos con su entorno, aunque cada implementación de nodo realizará internamente su propia función. Aquí pues se definen funciones tales como la conexión de los módulos entre sí (`operator` >> y <<) o la inclusión de un nuevo evento en la cola de eventos pendientes (`wakeUp`) aunque sin duda el método más importante y que más se usará en el código del simulador será `send`, a partir del cual se van a enviar los datos de un nodo a otro.

5.1.4. OTROS

Hay también otras dos clases que utilizaremos en el proyecto y que han sido facilitadas previo al desarrollo del simulador, se trata de las clases `distrib` y `lstat`.

Distrib es la clase encargada de definir distintos tipos de distribuciones, sus parámetros y valores de pico, necesario para caracterizar los distintos modelados así como otras variables. Las distribuciones caracterizadas son de tipo Constante, Uniforme, Exponencial, Normal, ErlangK, Pareto e Hiperexponencial.

Lstat es una clase que define a su vez otras dos que implementan operaciones básicas. Estas clases son `Counter`, `BasicStat` y `TempStat`, y algunas de éstas operaciones básicas son comparar valores, incrementar y decrementar variables,etc.

5.2. ANÁLISIS PREVIO

Basándonos de nuevo en la figura 5 del punto 5.1 del documento, y en el razonamiento posterior, que separaba el envío de datos en tres bloques fundamentales: nodos, paquetes e intercambio de información, vamos a analizar la necesidad de nuevas clases que modelen el estándar TCP.

Son los niveles inferiores los que especifican cómo se realiza el intercambio de información a través de la red, de origen a destino, por tanto el nivel TCP no añade nada al código previo del simulador encargado de esta tarea, por lo que se usa sus mismos métodos definidos en las clases scheduler, module y timer.

En cuanto a los paquetes de datos, se usará igualmente la clase packet definida en el código base del simulador, aunque con los cambios y añadidos ya mencionados.

Sin embargo, aunque los nodos intermedios y el medio físico sean modelados mediante las mismas clases comentadas en el punto anterior, los extremos de la conexión sí que deben ser distintos. Las diferencias entre la situación previa y la actual, así como la relación de herencia entre ellas se ilustra en la figura 6.

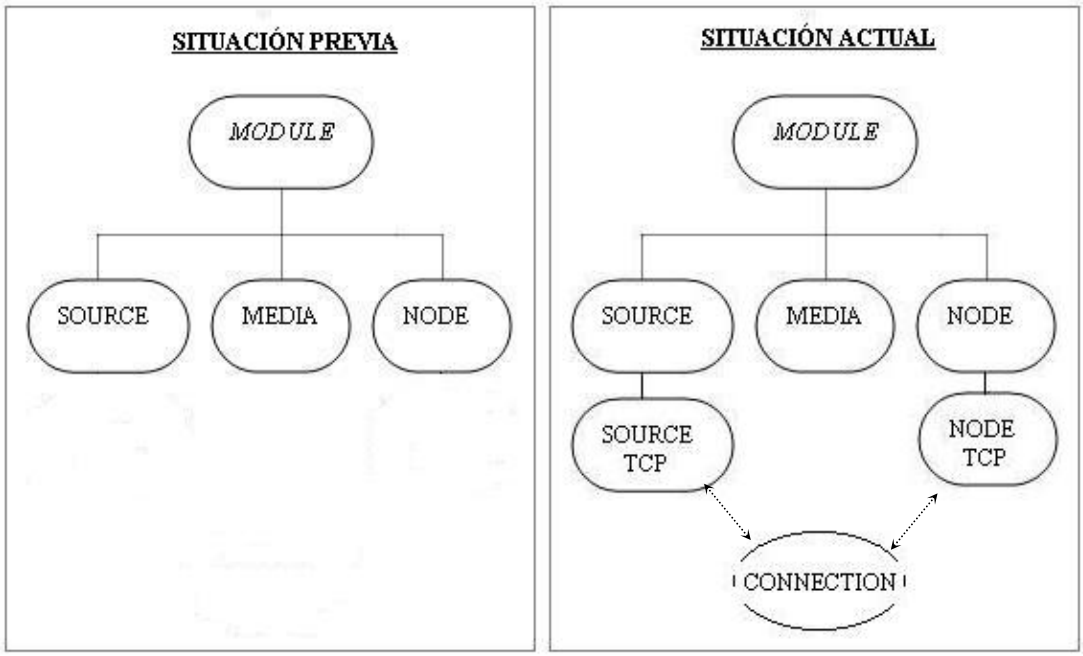

Figura 6.Relación entre clases en los extremos de la conexión.

La fuente de los datos TCP debe basarse sin duda alguna en la clase source, ya que da comienzo al envío de datos y debe enviarlos al destino cada intervalo de tiempo y estas funcionalidades ya las ofrece dicha clase. Sin embargo ya hemos comentado que hay que modificar los datos enviados, no serán directamente los que llegan de capas superiores sino que se envían almacenados en segmentos. Otro de los motivos para desarrollar una nueva clase es que source no recibe datos, sin embargo la fuente TCP debe hacerlo puesto que una de las características más importantes definida en el estándar es que la comunicación puede ser en ambos sentidos. Hay que definir por tanto un método que procese los segmentos de entrada y actúe en consecuencia. **SourceTCP** será la clase encargada de modelar una fuente de datos TCP.

El destino de los datos TCP será una nueva clase basada en node, ya que no es más que un tipo especial de nodo donde llega la información enviada por la fuente. Es necesario definir una nueva clase por dos motivos fundamentales: hay que procesar la información de entrada y actuar en consecuencia; y hay que enviar datos hacia la fuente para modelar la comunicación full-duplex. **SinkTCP** será la clase encargada de modelar el destino de los datos TCP.

Fuera de este esquema, es indispensable definir una nueva clase que gestione todos aspectos relacionados con la conexión en ambos sentidos, como el paso por los diferentes estados posibles dentro de una conexión, la gestión de los números de secuencia y asentimiento, o el manejo de la ventana. **Connection** será la clase encargada de modelar la conexión TCP.

Por último se define la clase **main**, donde, en líneas generales, se establece el esquema de conexión entre los nodos, se da valor al constructor de cada uno de éstos y se marca el comienzo y fin de la simulación.

En los siguientes puntos se analizarán en detalle el comportamiento de los métodos de cada una de estas clases para implementar cada característica que ofrece el nivel TCP. El código completo de cada una de ellas se encuentra en el segundo anexo de la memoria.

5.2.1. MAIN

La clase main contiene sólo del método main. En él se configura el modelo de conexión y se establece el comienzo y fin de la simulación.

En primer lugar el método main crea una instancia de la clase scheduler, que le va a permitir iniciar la simulación haciendo uso del método run que ya mencionamos en el punto 5.1.3 (intercambio de información).

Acto seguido verifica que los argumentos pasados por línea de comandos sean los adecuados, sacando un mensaje de error al usuario en caso contrario. Dicho mensaje informa al usuario del número correcto de parámetros que hay que pasar así como su significado. El modo en que hay que utilizar el simulador es:

```
main.exe <sem> <dur> <tll> <tser> <tam> <rc> <w_src> <w_snk>
[<q>][<ts_src>][<ts_snk>]
```

<sem> : semilla de la simulación. Número al azar que se utiliza para calcular los diversos estadísticos y parámetros de las funciones estadísticas necesarias para el simulador, por ejemplo para sacar los parámetros de la distribución de Pareto.

<dur> : duración de la simulación en segundos.

<tll > y <tser> : tll es el tiempo entre llegadas de paquetes (y el parámetro m de la distribución de Pareto) y tserv es el tiempo de servicio (y el parámetro alfa de la distribución de Pareto).

<tam> : tamaño de los paquetes de datos de usuario en octetos.

<rc> : régimen en células por tiempo de los enlaces.

<w_src> y <w_snk> : tamaño en octetos de las ventanas de recepción para la fuente y el destino de la comunicación respectivamente.

<q> : Parámetro opcional que define el tamaño de la cola de entrada. En caso de no darle valor se considerará cola infinita.

<ts_src> y <ts_snk> : Parámetro opcional que define el tamaño en octetos del segmento de datos en la fuente y el destino respectivamente. En caso de no pasarlo en la línea de comandos, el simulador considera por defecto que el tamaño de los segmentos en la fuente es 8500 octetos y en el destino es de 5000 octetos.

Además del uso del simulador también se informa al usuario de que en caso de fijar sólo uno de los tamaños de segmentos de datos, se considerará que es el de la fuente, ya que el compilador va tomando los argumentos en orden y el tamaño de segmentos en la fuente está definido antes que el tamaño de segmentos en el destino.

Después se comprueba que ni el tamaño de la ventana para el destino de datos sea cero, ni que el tamaño de segmento para la fuente sea cero, puesto que en ambos casos no se permitiría el envío de datos por parte de la fuente y no se llevaría a cabo la simulación. Se informa al usuario mediante un mensaje de error por pantalla si ocurriera alguno de estos casos.

Si el número de argumentos pasado por línea de comandos es el adecuado continúa la simulación estableciendo la semilla y definiendo los distintos nodos que constituyen el modelo de conexión por el que viajarán los datos y el sentido de dicha comunicación. Esta conexión viene determinada por el operador >> ó <<, según el sentido de transferencia de datos, métodos ambos de la clase module, que provocan que el tipo de module (node o source) que está en uno de los lados del operador acepte conexiones del otro lado, además de añadirlo a la lista de módulos por debajo o por encima.

En un primer momento se pensó como primera y mejor opción un esquema de conexión como el ilustrado en la figura 7, en el que entre los extremos de conexión y el medio físico hubiera dos nodos, uno que aceptara los datos del extremo y los pasara al medio físico, y otro que hiciera el trabajo inverso.

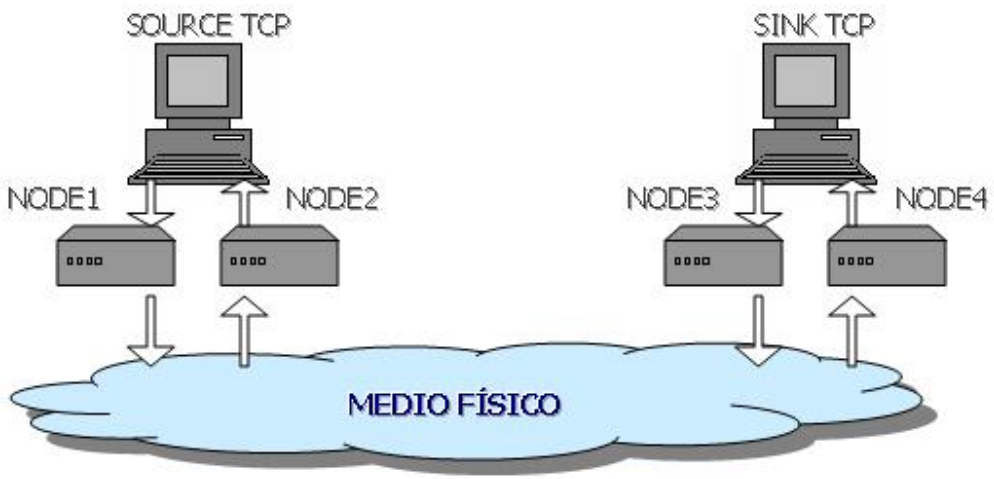

Figura 7. Primer modelo de conexión.

Sin embargo este esquema ha resultado ser incompatible con el funcionamiento básico del esquema inicial del simulador, debido tanto a la forma en como éste identifica

una conexión, como al modo de operación del método send de la clase module, encargado del envío de paquetes entre nodos.

Los identificadores de la conexión son asignados mediante los operadores << y >>, de forma que para una conexión como la descrita en la figura 7 las sentencias serían de la siguiente forma:

I. SourceTCP>>Node1>>Media<<Node4<<SinkTCP;
II. SinkTCP>>Node3>>Media<<Node2<<SourceTCP;

De lo anterior se desprende que:

- Si SourceTCP tiene, como es el caso, un identificador de conexión cero (connId_=0), provoca que todos los demás nodos tengan también el mismo identificador de conexión cero.
- De I. obtenemos además que en la lista de módulos por debajo de Node1 estará SourceTCP y de Media estará Node1; mientras que en la lista de módulos por encima de Node4 estará Media y de SinkTCP estará Node4.

- De II. obtenemos además que en la lista de módulos por debajo de Node3 estará SinkTCP y de Media estará Node3; mientras que en la lista de módulos por encima de Node2 estará Media y de SourceTCP estará Node2.

Es importante hacer mención de este comportamiento para comprobar la inviabilidad del modelo pretendido a raíz del modo de operación del método send, el cual acepta como parámetros el momento en que se realiza el envío y el paquete de datos a enviar, pero no el destino de esos datos. Por tanto, para llevar a cabo el envío al siguiente nodo la filosofía que sigue es, en resumen, que para un mismo identificador de conexión comprueba si los datos van hacia arriba o hacia abajo y, según el caso comprueba en la lista de módulos por encima o por debajo respectivamente los nodos que tiene con ese identificador y manda los datos a todos ellos.

En nuestro caso el conflicto surge en el método media: si la comunicación se originase de sourceTCP a sinkTCP, media debería enviar los datos sólo a node4 y sin embargo, cómo envía a todos los que estén en su lista de módulos por encima con el mismo identificador de conexión los envía también a node2 y vuelven a llegar a Source, esto es provoca una especie de eco. De igual forma ocurre cuando el envío de los datos se produce en sentido contrario, media debería enviar sólo a node2 y sin embargo los envía tanto a node2 como a node4.

Finalmente se optó por el modelo ilustrado en la figura 8, más sencillo pero que no presenta el problema anterior.

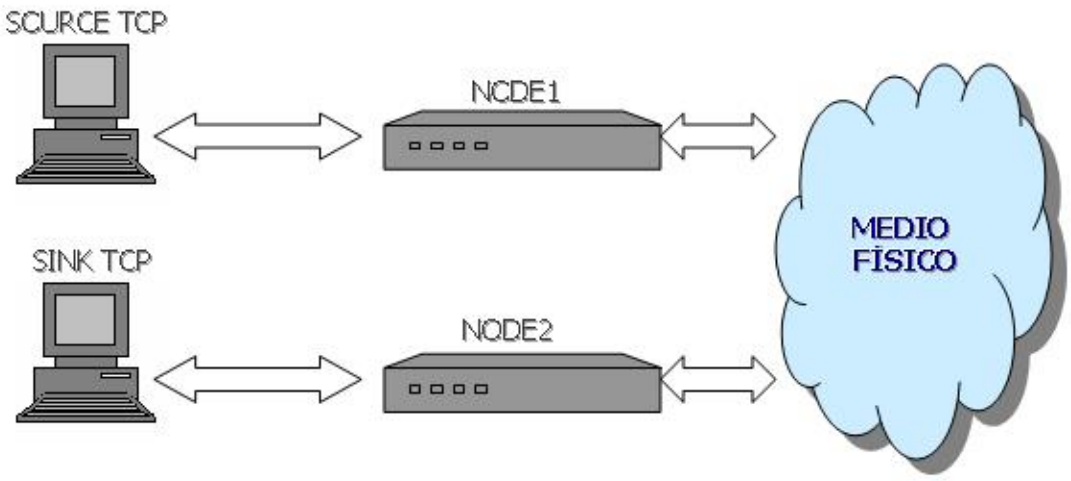

Figura 8. Modelo del conexión para el simulador.

En un primer lugar hay que definir estos nodos, creando una instancia para cada uno de ellos.

Para crear una instancia de la clase SourceTCP hay que pasarle al constructor los parámetros de la distribución de Pareto, ya que éstos son los parámetros que acepta la clase source y SourceTCP define en su constructor una instancia de dicha clase; el tamaño de la ventana para la fuente; y el tamaño del segmento (el valor pasado en la línea de comandos o bien el valor por defecto en caso de que no se haya especificado ninguno).

Para crear las instancias de la clase node se les pasa al constructor el régimen binario definido por el usuario; una instancia de la clase queue al que se le pasa el tamaño del la cola de paquetes (el valor pasado en la línea de comandos o bien el valor por defecto en caso de que no se haya especificado ninguno); y el número de servidores, igual a uno.

Para crear la instancia de la clase SinkTCP hay que pasar al constructor los mismos datos que para crear la instancia de node, ya que define una en el constructor; además hay que pasarle el tamaño de la ventana para el destino de los datos; y el tamaño del segmento (el valor pasado en la línea de comandos o bien el valor por defecto en caso de que no se haya especificado ninguno).

Por último, para crear la instancia de la clase media sólo hay que pasar al constructor el régimen binario.

Las sentencias para modelar la conexión especificada en la figura 8 serían:

```
sourceTCP>>node1>>medio<<node2<<sinkTCP
sinkTCP>>node2>>medio<<node1<<sourceTCP
```

Una vez definido el modelo, se establece el principio y final de la simulación, invocando a los métodos `start` y `stop` de la clase scheduler, haciendo uso de la

instancia definida al principio del main. Damos comienzo a la simulación mediante el método `run` de la misma clase.

Por último se sacan por pantalla algunos estadísticos de la simulación, por ejemplo el número de paquetes recibidos y perdidos por algún nodo.

5.2.2. SOURCE TCP

SourceTCP es la clase del simulador que modela la fuente de datos TCP. Su misión principal es iniciar tanto el establecimiento de la conexión, como su liberación y la transferencia de datos. El modo de actuación de la fuente es enviar segmentos de datos una vez que el segmento esté completo o se haya recibido una función push del usuario de capa superior. Los segmentos que recibe los procesa y actúa en consecuencia, eliminándolos finalmente.

Para ejecutar todas estas funcionalidades se definen una serie de métodos y variables, que son:

`start` : método que sobrecarga el método start de la clase source para evitar comenzar inmediatamente con el envío de datos desde la fuente, permitiendo así realizar una simulación de establecimiento de llamada.

`stop` : método que sobrecarga el método stop de la clase source evitando así que finalice el envío de datos inmediatamente, permitiendo de este modo realizar una liberación ordenada de la conexión.

`recvDown` : Procesa fundamentalmente los segmentos en recepción, y actúa en consecuencia. Desde este método se vuelve a llamar al método start de source una vez que está establecida la conexión, se aborta la conexión en caso de recibir un reset o se mantiene el control de la ventana de recepción entre otras funcionalidades.

`retxon` : método desde el que se reenvía un determinado segmento en caso de que haya expirado el contador de retransmisión y no se haya recibido asentimiento positivo del mismo.

stopSourceTCP : método que inicia la liberación de la conexión enviando al destino de los datos un segmento con el bit fin de la cabecera activo.

stateClose : establece el estado de la conexión Closed a partir de Time_Wait pasados 2MSL (vida máxima del segmento).

`isfull` : método que va llenando el segmento con los datos de usuario hasta que éstos completan el tamaño del segmento y da luz verde al envío desde la fuente. Si con el envío nos quedamos sin posiciones en la ventana de transmisión, detiene el siguiente envío hasta que volvamos a tenerlas.

fillPacket : método sobrecargado del source que introduce el valor de los datos para el paquete TCP.

TimeOut : cuando llega un segmento válido con el bit ack activo, elimina los segmentos asentidos de la cola de retransmisión. Cuando vence un contador de retransmisión, elimina también todos aquellos segmentos de la cola que tengan un número de secuencia mayor que el segmento a retransmitir.

En cuanto a variables de instancia, las más significativas son: `connIns_`, instancia de la clase connection que se encarga de la gestión de la conexión; `segsize_`, entero que indica en cada instante el tamaño que va teniendo el segmento de datos; `timerTCP_`, lista de elementos Timer, cada vez que se produce un envío de datos estando la conexión establecida se activa un timer del evento y se incluye en la lista; `firstsend_`, flag que controla el primer momento en que la conexión desde la fuente se encuentra establecida, es entonces cuando comienza el envío de datos; y por último `stopSend_`, flag que a true en caso de parar la fuente el envío de datos al quedarnos sin posiciones en la ventana, cuando se recibe un segmento válido con el campo de asentimiento significativo cambia su valor para que se proceda al envío de nuevo.

5.2.3. SINK TCP

La clase sinkTCP del simulador modela el destino o sumidero de los datos. Su principal misión es recibir los segmentos desde la fuente y procesarlos, actuando en consecuencia. Durante el establecimiento y liberación de la llamada, y durante la transferencia de datos al tratarse de conexión full-duplex, el destino debe tener la capacidad de enviar información a la fuente. El modo de proceder de sinkTCP es reenviar cada segmento de datos que recibe de la fuente una vez que la conexión se encuentra en estado Established, por supuesto con las propiedades definidas para los segmentos de sink, cambiándo el sentido del paquete de datos. De esta forma logramos modelar la comunicación en ambos sentidos.

Los métodos definidos en la clase sinkTCP son:

`RecvDown` : método sobrecargado de la clase node que recibe los segmentos de datos, y desde el que se llama al método original una vez que se procede al envío de cualquier dato. Desde el se mantiene el control del temporizador y la gestión de la ventana de recepción entre otras funcionalidades.

`sndClose` : método que simula una llamada de usuario Close cuando el nodo recibe una petición de liberación de conexión desde la fuente.

`retxon`: tiene exactamente la misma función que su homóloga en sourceTCP pero para sinkTCP, método desde el que se reenvía un determinado segmento en caso de que

haya expirado el contador de retransmisión y no se haya recibido asentimiento positivo del mismo.

isfull : 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.

Por su parte las variables de instancia que intervienen en la clase sinkTCP son: `connIns_`, instancia de la clase connection que se encarga de los parámetros de la conexión; `timerTCP_`, lista de elementos Timer, cada vez que se produce un envío de datos estando la conexión establecida se activa un timer del evento y se incluye en la lista; y finalmente `firstsend_`, flag que controla el primer momento en que la conexión desde la fuente se encuentra establecida, es entonces cuando comienza el envío de datos.

5.2.4. CONNECTION

Por último la clase connection del simulador modela todas las gestiones relativas a la conexión. Sus misiones fundamentales son implementar la máquina de estados mediante la cual la conexión irá pasando de un estado a otro durante las fases de establecimiento y liberación; y mantener el control y gestionar la actualización en cada instante de los números de secuencia y asentimiento, y de la ventana en recepción. Aunque también se encarga de simular las llamadas de usuario Open y Close, y de labores como la verificación de los segmentos de entrada.

Vemos cuáles son los métodos que define la clase connection:

`activeOpen` : método que simula una llamada de usuario Open activa para sourceTCP.

`passiveOpen` : método que simula una llamada de usuario Open pasiva para sinkTCP.

`CLOSE` : actualiza el estado de la conexión a Fin_Wait_1 y los valores del segmento a enviar.

`putClosed` : establece el estado de la conexión a Close.

`isconnEstab, isconnClWait, isconnTmWait` : métodos que permiten a otras clases saber si la conexión se encuentra en estado Established, Close_Wait o Time_Wait respectivamente.

`PackProcessing` : Máquina de estados. Cuando se recibe un segmento en un módulo TCP, el método comprueba si el segmento recibido es válido: debe ser coherente con el estado de la conexión en que nos encontremos y, en caso necesario, el número de secuencia y asentimiento deben ser los esperados. En este caso cambia si

fuera necesario el estado de la conexión. También comprueba el caso en que el segmento recibido contenga el mensaje reset.

`ControlPacket` : establece los valores adecuados a los bits de la cabecera del segmento TCP devuelto como parámetro del método.

`seqNumProcessing` : método que establece y actualiza según sea la naturaleza del segmento a enviar (de datos, de control o asentimiento) el número de secuencia en el segmento que se va a enviar.

`seqNumCheck` : método que comprueba que el número de secuencia del segmento entrante es el esperado en recepción, y en caso afirmativo actualiza dicho número.

`putAck` : método que devuelve al número de asentimiento actual, esto es, el siguiente número esperado en recepción.

`segmentSize` : devuelve el tamaño del segmento en octetos.

`winCalculate` : método encargado de mantener la ventana de recepción, calcula las posiciones de la ventana y modifica el valor, tanto de dicha ventana (en caso de congestion), como del tamaño del segmento en caso necesario.

`reset` : procesa un reset en recepción, actuando en consecuencia al estado en que se encuentre la conexión.

`abort` : método que informa al usuario si hay o no que abortar la conexión.

`ack` : informa al usuario si hay o no que mandar un asentimiento positivo al otro extremo.

`returnError` : comprueba si se ha producido un reset o hay que enviar un asentimiento debido a un error durante la etapa de comprobación del segmento.

`errorMsg` : se llama al método si 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. Si llega a sinkTCP una petición de abortar la conexión se aprovecha de las facilidades de este método para construir el segmento de petición de abortar a la fuente.

`wndLimit` : devuelve el número de octetos que aún se puede enviar al otro extremo hasta completar la ventana de transmisión.

`TxonWindow` : devuelve el valor real de la ventana de transmisión.

`putWindow` : actualiza el valor de la ventana de recepción con el valor recibido del otro extremo.

Las variables de instancia definidas en la clase connection son:
-connstate_, variable de tipo enumerado que almacena el estado en que se encuentra la conexión, puede tomar los valores CLOSED, LISTEN, SYN_SENT, SYN_RCVD, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSING, TIME_WAIT, CLOSE_WAIT y LAST_ACK .
-sndWnd_ y rcvWnd_ , enteros de tipo unsigned long que almacenan el tamaño en octetos de la ventana en transmisión y recepción respectivamente.
-sndNxt_, variable tipo entero unsigned long que especifica el siguiente número de secuencia que se va a enviar.
-rcvNxt_, variable tipo entero que almacena el siguiente número de secuencia esperado en recepción.
-sndUna_, entero de tipo unsigned long que indica el último número de secuencia asentido del transmisión.
-ssizeInicial_, tamaño del segmento pasado por el usuario por línea de comandos, o bien tomado por defecto en caso de no especificar.
-ssize_, entero que almacena el tamaño real del segmento en cada instante, en condiciones normales coincide con el valor de la variable ssizeInicial, pero puede variar como consecuencia del tamaño de la ventana en recepción.
-txonWnd_, tamaño de la ventana de transmisión en cada instante, dependiendo del estado de congestión.
-recuperación_, flag que indica cuando nos encontramos en un estado de recuperación de la ventana original después de un estado de congestión en la red.
-reset_, variable tipo booleano que indica al transmisor si se ha generado un reset como resultado de la comprobación del segmento entrante.
-abort_, variable de valor positivo en caso de que haya que abortar la conexión.
-asentimiento_, flag que indica si, encontrándose la conexión en un estado sincronizado, ha llegado un segmento no acorde con el mismo. Avisa al extremo implicado que debe mandar un asentimiento con los parámetros de envío actuales.

5.3. ESTABLECIMIENTO DE LA CONEXIÓN

Una vez que tenemos una visión general de los métodos y variables que componen cada una de las nuevas clases desarrolladas para el simulador del estándar TCP, veamos cómo a partir de ellas ofrecemos todas y cada una de las funcionalidades del Protocolo de Control de Transporte.

Empezamos analizando el modo en que simulamos un establecimiento de conexión entre la fuente y el destino de los datos antes de la etapa de intercambio de información entre ellos.

El procedimiento de establecimiento de conexión que implementa el simulador se denomina negociación en tres pasos (3-way handshake) y su modo de proceder se discutió en el segundo capítulo de la memoria, en el punto 2.2.3 (establecimiento de conexión). No existe la posibilidad por tanto de encontrarnos en un caso de establecimiento simultáneo de la conexión.

Cada uno de los pasos a seguir se encuentran ilustrados en la figura 9, basada en la figura 3 del mismo punto anterior, donde el TCPa será la fuente (sourceTCP) y el TCPb será el destino de la comunicación (sinkTCP). Se ha enumerado cada paso a seguir con el objeto de que resulte más fácil su seguimiento en el código.

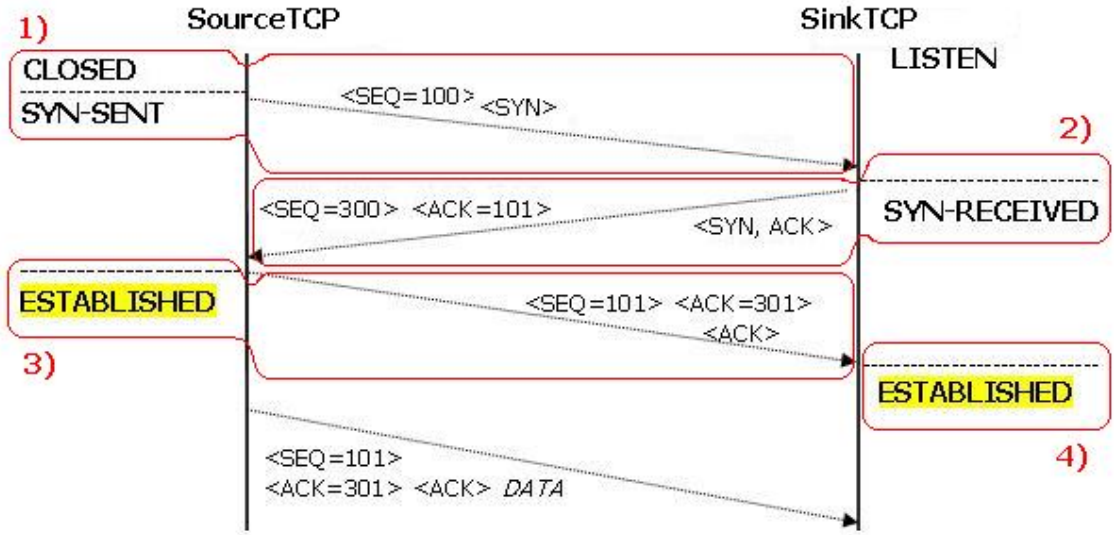

Figura 9. Pasos a seguir durante el establecimiento de conexión.

En el construtor de ambos extremos de la conexión se define una instancia de la clase connection, `connIns_`, mediante la que hacemos uso de sus facilidades para gestionar la conexión. Al definir la instancia, en el mismo constructor de connection, se establece el estado de la conexión a Closed. Vamos a detallar cada paso enumerado en la figura 9.

En primer lugar se debe simular una llamada de usuario Open pasiva para sinkTCP, que esperará a que el otro extremo sea el que comience una comunicación de datos. Se produce para ello una llamada al método passiveOpen de connection en el propio constructor de sinkTCP, el cual simplemente cambia el estado de la conexión de Closed a Listen, antes incluso de que comience la simulación.

1) Una vez iniciada la simulación, hay que evitar que la fuente comience directamente el envío de datos al otro extremo. Se consigue sobrecargando el método start de source. El nuevo método start de sourceTCP realiza una llamada al método activeOpen de connection, el cual:
-Cambia el estado de la conexión, pasando de Closed a Syn_sent .
-Prepara un segmento de control con el bit fin activo para enviarlo al otro extremo. Para establecer los datos de control de la cabecera TCP se invoca al método ControlPacket de connection indicándole que el bit syn debe estar activo, mientras que ack y fin deben permanecer a false.
- Se establece en el segmento y se actualiza el número de secuencia mediante el método seqNumProcessing, que veremos en detalle en el apartado dedicado a ellos más adelante en la memoria.

Por último en start se asigna el origen e identificador de la conexión y se envía a sink haciendo uso del método send de module.

2) Este segmento enviado llega al método recvDown de sinkTCP. A la hora de procesar un segmento de entrada, ambos extremos deben verificar en primer lugar que el segmento recibido no se trata de un mensaje especial reset o está dañado (tiene errores en el checksum).

Si esto es así, se comprueba si se ha producido algún cambio en la ventana de recepción originada en el otro extremo a causa de un estado de congestión en la red. Se debe comprobar antes de procesar el segmento de entrada ya que la ventana marca el límite para el número de secuencia del segmento entrante.

Se verifica a continuación que el segmento es válido en relación con el estado de la conexión en que se encuentra el nodo mediante una llamada al método PackProcessing de connection. Este método, tal y como se ha comentado en el punto anterior, se encarga de comprobar el segmento entrante y si éste es válido, actualiza el estado de la conexión y las variables asociadas a los espacios de número de secuencia y de asentimiento, así como de elaborar el segmento a enviar al otro extremo como respuesta al recibido. Estando en estado Listen sólo van a ser aceptables segmentos de petición de conexión, si tienen el bit SYN activo no hay que comprobar nada más.

En este caso, el segmento de entrada tiene el bit syn activo, por lo que una llamada a PackProcessing estando la conexión en estado Listen provocaría:
- La conexión pasa a Syn_Rcvd.
- Como se observa en la figura 9, sink debe enviar un segmento syn a source, asintiéndole a su vez el segmento recibido. Por tanto se deben activar los bits syn y ack de control, mediante el método ControlPacket, y también se incluye en el segmento el tamaño de la ventana de sink.
-Con una llamada al método seqNumProcessing se establece y actualiza el número de secuencia del paquete a enviar.
- Se actualiza el valor del siguiente número de secuencia esperado.
- Al ser el primer segmento recibido del otro extremo, actualizamos el tamaño de la ventana en transmisión con el valor indicado en el segmento de entrada.

El método devuelve como resultado el segmento a enviar o null si no hubiera necesidad o se hubiera producido error en la comprobación del segmento. En este caso el paquete devuelto es no nulo, con lo que se comprueba a continuación si la conexión se encuentra o no establecida. Como aún no lo está, ponemos en el segmento el número de asentimiento mediante el método de connection putAck y fijamos el origen e identificador de conexión para el segmento y lo enviamos al otro extremo.

3) El paquete de control enviado por sinkTCP llega al método recvDown de sourceTCP. Al igual que en sink se comprueba en primer lugar que no haya errores en el segmento, suponemos que no es el caso (veremos en detalle cómo se comporta el simulador frente a situaciones de error en el capítulo dedicado a ello).

Por la misma razón expuesta en el punto anterior se debe comprobar en primer lugar si se ha modificado la ventana de recepción debido a la detección de una situación de congestión en el extremo opuesto de la conexión, que reduce como consecuencia su ventana de transmisión. Si ha ocurrido dicha modificación, se actualiza el valor de la ventana de recepción en connection.

Se llama a continuación al método PackProcessing estando la conexión en estado Syn_sent. Como en la implementación del simulador ya hemos dicho que no está contemplada la iniciación de conexión simultánea, nos debe llegar del otro extremo un paquete SYN con el asentimiento del propio. Hay que verificar por tanto que el segmento tenga los bits de control syn y ack activos, y que el número de asentimiento sea el esperado. Si se produce todo esto:
- La conexión pasa de estado Syn_sent a Established, con lo que queda establecida.
- Se prepara el segmento de asentimiento a enviar como respuesta al recibido: mediante el método ControPacket creamos el paquete con el bit ack a true, y establecemos en el segmento el número de secuencia mediante seqNumProcessing, actualizando de este modo también el siguiente número de secuencia a enviar.
- Se modifica por último las variables de la conexión necesarias como consecuencia del segmento recibido: se actualiza el siguiente número de secuencia esperado en recepción, las ventanas de transmisión y el último número de secuencia asentido.

Si, como es el caso, el paquete devuelto por PackProcessing no es nulo, se comprueba si está la conexión o no en estado establecida. En este caso entramos en el bucle por primera vez.

Como hemos recibido un asentimiento se produce una llamada al método TimeOut, que se encarga de sacar de la lista de retransmisión los segmentos asentidos. El funcionamiento de este método lo veremos más adelante en el capítulo dedicado a ello.

Mediante winCalculate recalculamos en caso necesario el valor del tamaño del segmento (si éste es mayor que la ventana de transmisión) y, al ser la primera vez que entramos en estado establecido, mandamos el ack del syn al otro extremo, estableciendo el origen, identificador de conexión y número de asentimiento.

Por último se da comienzo al envío de datos llamando al método start de source.

4) Llega este paquete a recvDown de sinkTCP. Como no tiene errores se llama al método PackProcessing estando en estado Syn_received, estado en el que sólo se admitirán segmentos de asentimiento. En este caso se comprueba que tanto el número de secuencia como el número de asentimiento son los esperados. Si esto ocurre:
- La conexión pasa a Established.
- En respuesta a éste no se envía ningún segmento al otro extremo, por lo que no preparamos ningún segmento y se devolverá como resultado del método un paquete nulo.
- Se actualiza el último número de secuencia propio asentido.

Al devolver como resultado null, sólo quedaría comprobar si esto es así debido a que se ha producido un error durante la comprobación del segmento o si es que no hay nada que devolver como respuesta al otro extremo. Con tal fin se comprueba el resultado del método returnError de connection, que devuelve true si se ha producido un reset o hay que enviar al otro extremo un asentimiento por error ocurrido durante un estado sincronizado. No se realiza ninguna acción más al ser el resultado de la llamada al método negativo.

5.4. LIBERACIÓN DE LA CONEXIÓN

Al igual que el establecimiento de la conexión usa el procedimiento denominado negociación en tres pasos, la fase de finalización usa una negociación en cuatro pasos (4-way handshake), tal y como se comentó en el punto 2.2.4 del presente documento.

Aunque en el estándar está contemplada la posibilidad de que ambos TCPs inicien simultáneamente la liberación de la llamada, no es lo habitual, por tanto en el simulador comenzamos el proceso de finalización desde sólo un extremo, que es el modo normal de liberación. Puesto que es el método source el que tiene noción de cuándo se termina la simulación (el método stop se llama en dicho instante) y es el método sourceTCP el que hereda de esta clase, parece lógico elegir este extremo como punto de inicio para el proceso de liberación de la conexión.

Cada uno de los pasos a seguir se encuentran ilustrados en la figura 10, basada en la figura 4 del mencionado punto 2.2.4, donde el TCPa será la fuente (sourceTCP) y el TCPb será el destino de la comunicación (sinkTCP). Se ha enumerado cada paso a seguir con el objeto de que resulte más fácil su seguimiento en el código.

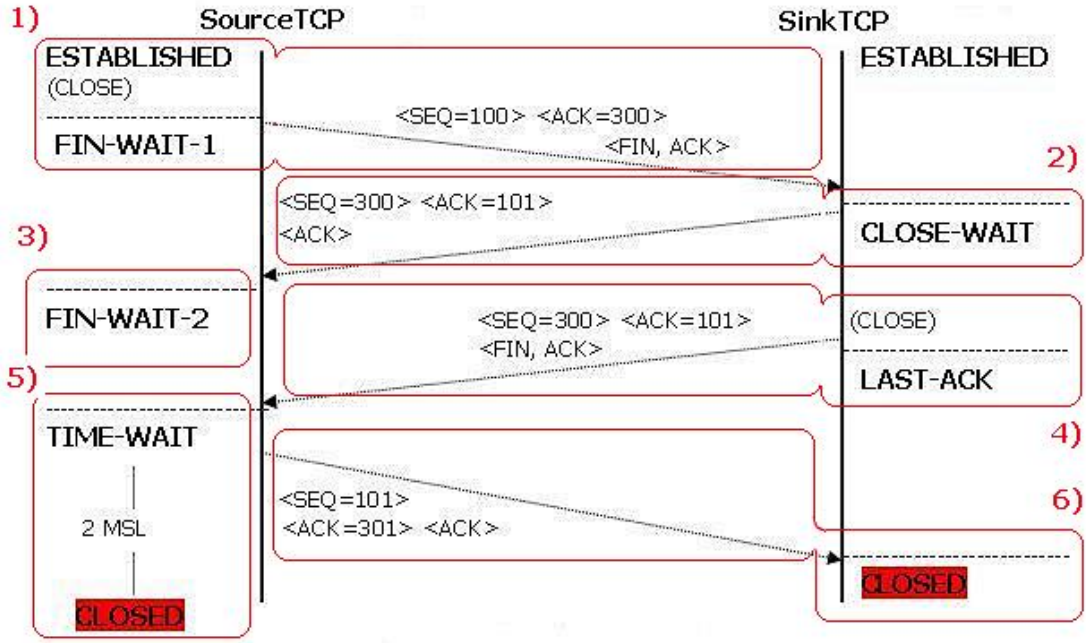

Figura 10: Pasos a seguir en la liberación de la conexión.

1) SourceTCP sobrecarga como hemos comentado el método stop de source para evitar que finalice la simulación de forma brusca. Desde el nuevo método stop se realiza una llamada al método stopSourceTCP en el instante determinado por el usuario como límite para la simulación y a continuación llama a stop de sourceTCP para que finalice el envío de segmentos de datos desde la fuente.

Desde stopSourceTCP se llama al método CLOSE de connection, que provoca:
-La conexión pasa de establecida a estado Fin_wait_1.
-Crea el mensaje a enviar al otro extremo haciendo uso del método ControlPacket, con los bits fin y ack activados, que será devuelto como resultado del método.
-Se cambia el valor del bit de cabecera psh a true. Ya se comentó en el punto dedicado a la liberación de memoria que todo mensaje fin lleva asociada una función psh.
-Se establece el número de secuencia apropiado en el segmento y se actualiza el siguiente número de secuencia a enviar mediante seqNumProcessing.
-Y se introduce en el segmento por último el número de asentimiento.

Ya de nuevo en stopSourceTCP se introduce el origen e identificador de conexión y se envía al otro extremo.

2) El segmento enviado llega al método recvDown de sinkTCP, donde, tras comprobar que el segmento no introduce ningún error ni ha habido cambios en la ventana de recepción, se produce la llamada al método PackProcessing estando la conexión para sink establecida.

Estando en estado establecido lo único inaceptable sería un segmento syn, implicaría que se ha llegado un segmento erróneo o de una conexión antigua, lo cual es imposible en el simulador. Hay que comprobar, por tanto, que el segmento no tenga activo este bit de cabecera y que los números de secuencia y asentimiento recibidos sean los adecuados. Se discutirá acerca de las comparaciones necesarias en el punto dedicado a la comunicación de datos. Durante el proceso, si el segmento resulta válido, se actualiza el siguiente número adecuado en recepción, que será el asentimiento que pongamos en el segmento a enviar actual.

Una vez validado el segmento recibido, se actualiza el último número de secuencia asentido y se crea un segmento de asentimiento (con el bit ack de cabecera activo) mediante ControPacket, y se comprueba si el bit fin del paquete recibido está activo, como es el caso, lo que provoca:
-El estado de la conexión pasa de Established a Close_Wait.
-Como realmente scheduler deja de lanzar eventos en el momento en que finaliza la simulación, habrá con total seguridad segmentos que hayan sido enviados pero que se habrán quedado por el camino. Esta finalización tan brusca provoca que los números de secuencia o de asentimiento no estén totalmente sincronizados, lo cual es indispensable para intercambiar los segmentos necesarios durante la negociación en cuatro pasos. Lo que se hace es “forzar” a que los números dentro del espacio de secuencia sean los necesarios, se fuerza así tanto el número de secuencia del segmento (que debe ser el de asentimiento recibido) como el número de asentimiento (uno más que el recibido).
-Por último ponemos en el segmento el número de secuencia y actualizamos el siguiente a enviar mediante seqNumProcessing.

Ya de nuevo en recvDown, comprobamos que el nodo no se encuentra ya en estado establecido, con lo que pasamos a comprobar a su vez si el nuevo estado de la conexión es Close-wait. Al serlo, se llama al método sndCLOSE de sinkTCP en un instante posterior al actual de forma que a sink le de tiempo a asentir el fin recibido antes de enviar el propio.

Por último se fija el origen, identificador de conexión y número de asentimiento del segmento de asentimiento devuelto por PackProcessing y se envía al otro extremo.

3) El segmento enviado por sink llega a sourceTCP a través de su método recvDown. Una vez comprobado que no se tiene errores y no se ha modificado la ventana de recepción, se llama a PackProcessing en estado Fin_wait_1. Como el nodo a enviado un fin, está esperando que le sea asentido, por lo que sólo se aceptarán segmentos que tengan el bit ack de cabecera activo. Se comprueba también, por supuesto, que tanto el número de secuencia como el de asentimiento son los esperados. Como todo esto ocurre cambia el estado de la conexión a Fin_wait_2, y se actualiza el siguiente número de secuencia esperado y el último propio asentido.

Como se puede observar en la figura 10, no se envía ningún segmento como respuesta al recibido, por lo que el método PackProcessing devuelve null como resultado. No se realiza ninguna acción más, ya que el null no es debido a un error durante la validación del segmento.

4) Estamos ahora en el método sndCLOSE de sinkTCP que fue llamado al final del paso 2 al estar el nodo en estado CLose-wait. Este método es el homólogo de stopSourceTCP en dicho extremo, su función es la misma. Desde sndCLOSE se llama al método CLOSE de connection, que provoca:
-La conexión pasa de Close_Wait a estado Last_ack.
-Crea el mensaje a enviar al otro extremo haciendo uso del método ControlPacket, con los bits fin y ack activados, que será devuelto como resultado del método.
-Se cambia el valor del bit de cabecera psh a true (todo mensaje fin lleva asociada una función psh).
-Se establece el número de secuencia apropiado en el segmento y se actualiza el siguiente número de secuencia a enviar mediante seqNumProcessing.
-Y se introduce en el segmento por último el número de asentimiento.

Ya de nuevo en sndCLOSE se introduce el origen e identificador de conexión y se envía al otro extremo.

5) Llega el semento del fin a recvDown de sourceTCP, donde llamamos a PackProcessing tras comprobar nuevamente que el segmento no tiene errores y no se ha modificado su ventana de recepción.

En estado Fin_wait_2 nos encontramos tras mandar un mensaje fin, por lo que serán aceptables segmentos con el bit fin activo y con el ack activo. Se comprueba también que los números de secuencia y asentimiento son los esperados, siendo esto así:
-El estado de la conexión pasa a ser Time_Wait.

-Se prepara el segmento de asentimiento a enviar al otro extremo como respuesta al recibido (ver figura 10), creando mediante ControlPacket un segmento con el bit de cabecera ack activo e introduciendo en el mismo el número de secuencia mediante seqNumProcessing, que actualiza además el siguiente número de secuencia a enviar.
-Se actualizan el siguiente número de secuencia esperado (el número de asentimiento a enviar en el paquete) y el último número se secuencia asentido que será uno menos que el número de asentimiento recibido.

Como el segmento devuelto por PackProcessing no es nulo y además la conexión no se encuentra en estado Established, se pasa a comprobar si el estado de la conexión es Time_Wait, llamando en tal caso al método stateClose de sourceTCP en el instante actual mas dos MSL (longitud máxima de segmento) tal y como indica el estándar, cuyo único cometido es establecer definitivamente el estado de la conexión a Closed para sourceTCP, dando por concluida la llamada desde este extremo de la conexión.

Tras la llamada a stateClose, se introduce en el segmento de asentimiento devuelto por PackProcessing el origen, identificador de la conexión y número de ack y se envía a sink.

6) Llega por último el segmento de asentimiento a sink, se procesa en recvDown donde, tras comprobar que no tiene errores ni le informa de cambio en su ventana de recepción, pasamos al método PackProcessing con Last_Ack como estado de conexión. El modo de actuar es análogo a cuando se llamaba al método con un estado de conexión Fin_wait_1, al igual que entonces sólo serán válidos segmentos con el bit ack activo ya que estamos esperando el asentimiento del fin enviado. Se comprueba también la validez del número de secuencia y asentimiento recibido. Al ocurrir todo esto, el estado de la conexión pasa a ser Closed, y se actualiza el siguiente número de secuencia esperado y el último asentido.
No se envía ningún segmento como respuesta al recibido, por lo que el método PackProcessing devuelve null como resultado. No se realiza ninguna acción más, ya que el null no es debido a un error durante la validación del segmento.

Con esto se concluye la fase de liberación de conexión, estando ambos extremos en estado Closed.

5.5. COMUNICACIÓN DE DATOS : FIABILIDAD

Una vez que la conexión está establecida, comienza la transferencia de datos en ambos sentidos mediante el intercambio de segmentos.

La propiedad más importante de la comunicación de datos en TCP es que ésta se realiza en modo full-duplex y por tanto el simulador debe implementar esta opción.

La idea para obtener comunicación en ambos sentidos de la conexión se basa en dos conceptos: uno de ellos es que la fuente es la que emite datos y el destino, en lugar de desechar dichos datos los reenvía a la fuente con sus propias características de longitud, número de secuencia, etc. El otro concepto es permitir al usuario que sea él el

que elija el modo de funcionamiento del TCP, dándole la opción de pasar como parámetro en la línea de comando el tamaño de segmento para source y para sink. No se va a permitir pasar cero como longitud de segmento a la fuente, ya que este comportamiento carece de lógica, pero sí que se podrá pasar longitud de segmento cero para el destino de los datos, forzando de este modo a que el destino no envíe datos, sino que se limite a asentir los del origen.

También se señala en el estándar que el envío de los datos de usuario al TCP de destino se realiza en propia conveniencia del TCP origen cuando éste llena su segmento. Basándonos en el comportamiento del simulador base comentado, resulta necesario como ya se señaló, la implementación de dos nuevos métodos en source (uno de ellos sobrecargado en sourceTCP) para evitar que el envío de los datos de usuario se realice de forma automática y para caracterizar el modelo de envío que define TCP.

El primero de ellos, putPsh, provoca que cuando llegan datos de usuario se pueda establecer el valor del bit de cabecera psh, en este código por defecto a false, antes de comprobar si el segmento está listo o no para el envío. Al ser un método de source, permitimos que cualquier otra clase que herede de ella, implementando por ejemplo un usuario de capa superior, pueda sobrecargar el método poniendo el valor del bit psh a su conveniencia. La función push se define con el propósito de que el usuario de capas superiores tenga la seguridad de que se transmiten todos los datos que han mandado al otro extremo.

Esta función implica que los datos de usuario entregados al TCP transmisor hasta ese punto deben ser enviados inmediatamente al otro extremo aún cuando no se haya completado el segmento. Ésta especificación del estándar no afecta para nada a la implementación del simulador, ya que debe ser dada al mismo por los niveles superiores. Sin embargo, cuando se especifica una función push, se debe activar el flag psh en la cabecera, para que de este modo se “empujen” en el destino inmediatamente los datos al usuario receptor sin esperar a que el buffer esté lleno. Esta característica es la que modelamos mediante el método putPsh al estar especificada en el estándar, de modo que se comprobará que el bit psh de cabecera esté activo en la fuente de datos a la hora de determinar si se envía el segmento o no.

El otro nuevo método definido en source es isfull. Como ya se comentó en el punto 5.1.1. simplemente devuelve true si el segmento está listo para enviarse, por lo que se debe llamar siempre antes de proceder al envío del segmento desde source. Este método se sobrecarga en sourceTCP, de este modo no se varía el comportamiento del simulador básico forzando a que el método devuelva en source siempre true.

Cada vez que se produce una llamada al método isfull se calcula el número de octetos libres hasta completar la ventana de transmisión como uno menos que el resultado de la llamada al método wndLimit de connection, que resta el valor del siguiente número de secuencia a enviar a la suma del último número de secuencia asentido más la ventana de transmisión; y se añade al tamaño del buffer de datos el tamaño del paquete de datos de usuario.

Se enviará el segmento sólo en el caso de que hayamos completado un segmento de datos, nos quedemos sin posiciones en la ventana o se nos haya indicado una función push. Se devuelve false si no ha ocurrido ninguna de estas condiciones. En el caso de que enviemos puede ocurrir:

- Si hemos llegado al límite de la ventana, el tamaño del segmento será el número de octetos libres hasta completar la ventana calculado antes, calculamos el nuevo tamaño del buffer y paramos el envío de datos por parte de la fuente hasta recibir algún asentimiento. Se establece para ello el valor del flag que controla el envío a false y se llama al método stopSource de source.

- Si hemos completado el segmento, el tamaño del segmento será el tamaño máximo de segmento fijado por el usuario del simulador, o por la ventana de transmisión en caso de ser menor, y recalculamos el tamaño del buffer de datos.

- Si por el contrario se nos ha indicado una función push, el tamaño del segmento será el tamaño actual del buffer y actualizamos el nuevo tamaño a cero.

Cuando se da luz verde al envío de datos, hay que dar valor a cada uno de los campos del segmento a enviar. Source define para esto el método fillPacket, que sourceTCP sobrecarga para introducir los datos específicos del paquete TCP.

Al llenar el segmento TCP se da valor a cada uno de los bits de cabecera, activando sólo el bit de asentimiento; se establece el número de secuencia del segmento mediante seqNumProcessing, con lo que se actualiza el siguiente número de secuencia a enviar; y se establece también el número de asentimiento mediante putAck. Como ya veremos en detalle en el punto dedicado al contador de retransmisión, ahora que se va a proceder al envío de un segmento de datos, se activa el temporizador asociado a dicho segmento.

Una vez completada la información del segmento se envía desde source el segmento de datos a sink.

El segmento recibido en sinkTCP se analiza para ver si presenta algún tipo de error o indica un cambio en la ventana de recepción, de no ser así se llama a PackProcessing de connection.

Con el estado de conexión establecido sería inaceptable que me llegara un segmento con el bit syn activo como ya hemos comentado con anterioridad, una vez comprobado que esto no sucede y que los números de secuencia y asentimiento son los esperados, se actualiza el último número de secuencia asentido y se devuelve como resultado de la llamada un segmento de asentimiento construido mediante ControlPacket.

Como el resultado del método no es null y estamos en estado establecido, realizamos una llamada al método timeOut, que veremos en detalle el punto dedicado al contador de retransmisión, para que elimine de la lista todos aquellos segmentos asentidos con el segmento recibido y desactive también su temporizador. A continuación

se realiza una llamada a winCalculate de connection, cuyo funcionamiento también será detallado más adelante, en el punto dedicado al control de flujo, pero que fundamentalmente lo que persigue es ir actualizando la ventana de transmisión si nos encontramos en un estado de recuperación tras uno de congestión en la red, en el que se disminuyó el tamaño de la misma.

Comprobamos entonces en sinkTCP el resultado de la llamada al método isfull. El concepto de este método es análogo al de sourceTCP, pero evidentemente no puede ser igual, ya que aquí entramos tras recibir un asentimiento. Su comportamiento es enviar siempre que tengamos octetos libres en la ventana de transmisión, distinguiendo el caso de si hemos llegado al límite de la ventana antes de llenar completamente el segmento en cuyo caso el tamaño del mismo será el número de octetos libres en la ventana.

Si el resultado de la llamada a isfull resulta afirmativo, se establece en el segmento el número de secuencia y se asentimiento, actualizando además el siguiente número de secuencia a enviar, se fija el origen y el identificador de la conexión, y se envía a sourceTCP.

El segmento llega a recvDown de sourceTCP. Los primeros pasos a seguir son idénticos a los seguidos en sinkTCP. Empiezan a diferir en el instante en que se comprueba que el resultado de la llamada a PackProcessing no es null y el estado está establecido, tras las sendas llamadas a los métodos TimeOut y WinCalculate. Mientras que en sinkTCP había que preocuparse en construir el segmento de datos hacia el otro extremo, aquí en sourceTCP no es necesario, el envío y la recepción están completamente independizados. Por tanto aquí lo único que será necesario es comprobar si el envío de datos está parado (nos lo indica isfull cuando envía un segmento porque ha llegado al límite de la ventana) para volverlo a activar ya que hemos recibido un asentimiento. Lo reactivamos mediante una llamada al método de startSource de source.

5.5.1. NÚMEROS DE SECUENCIA Y ASENTIMIENTO

Con el objetivo de asegurar la fiabilidad en la comunicación de datos TCP debe recuperar datos que se pierden, dañan, duplican o entregan fuera de orden. Parte de esta funcionalidad se consigue asignando un número de secuencia a cada octeto transmitido, y requiriendo un asentimiento positivo (ack) del TCP en recepción.

El tratamiento de los números de secuencia y asentimiento se encuentra en la clase connection, y se gestionan fundamentalmente actualizando el siguiente número de secuencia a enviar, el siguiente número de secuencia esperado y el último asentido.

Mediante el método `seqNumProcessing` se establece el número de secuencia en el segmento a enviar y se actualiza el siguiente a enviar. Cada vez que se realiza una llamada a este método, se establece en primer lugar el número de secuencia en el segmento pasado como parámetro del método, y luego dependiendo del tipo de segmento que sea se actualiza el siguiente a enviar. Si se trata de un paquete de control (bit syn o fin activos), el siguiente número de secuencia será uno más que el actual, mientras que si se trata de un segmento de datos, será el actual más la longitud del

segmento. Si se tratara de un paquete de asentimiento no se actualiza el siguiente número de secuencia a enviar, no varía ya que un asentimiento no consume posiciones en el espacio de secuencia.

La llamada al método seqNumProcessing se realiza todos aquellos momentos en los que se va a proceder al envío de segmentos que requieren número de secuencia. Durante la transferencia de datos la llamada se realiza fundamentalmente, en sourceTPC desde el método fillPacket si se envían datos o desde recvDown en caso de enviar sólo asentimientos, y en sinkTCP en el método recvDown tanto si se envían segmentos de datos como asentimientos.

El establecimiento del número de asentimiento en un segmento se realiza mediante `putAck`, que simplemente devuelve como resultado a su llamada el valor del siguiente número de secuencia que se espera en recepción, el cual coincide por definición con el número de asentimiento.

Durante todo el desarrollo del establecimiento y la liberación de la conexión, en la inmensa mayoría de los casos en que se llama al método PackProcessing, ha sido necesario comprobar la validez tanto del número de secuencia como del número de asentimiento. Esto es especialmente crítico durante el intercambio de información entre nodos, ya que durante la fase inicial y final de la conexión se esperan siempre unos valores determinados.

Las comparaciones en el número de asentimiento y de secuencia se han realizado conforme al estándar, comentadas en el punto 2.2.6 “Comunicación de datos” del presente documento. Cuando se realiza una llamada a PackProcessing estando la conexión establecida, se debe cumplir:

- En primer lugar que el número de asentimiento recibido sea mayor que el último número de secuencia asentido, y menor o igual que el siguiente a enviar. Si esto no ocurre es que ha llegado un segmento que no tiene nada que ver con la conexión actual. En este caso se avisa al usuario mediante un mensaje por pantalla de que ha ocurrido este error, no se devuelve nada como resultado de la llamada al método y se ignora.

- Para comprobar la validez del número de secuencia durante la etapa de transferencia de datos, se define un nuevo método, `seqNumCheck`. El segmento será válido si el resultado de su llamada es positivo. Cada vez que seqNumCheck es llamado, comprueba en primer lugar la longitud del segmento pasado como parámetro. Si es cero, puede ocurrir que el tamaño de la ventana de recepción sea también cero, con lo que el segmento será válido si el número de secuencia recibido es realmente el que se esperaba; o puede ocurrir que el tamaño de la ventana sea mayor que cero, con lo que será válido si el número de secuencia recibido es mayor o igual que el siguiente esperado y menor que la suma del siguiente esperado mas la ventana de recepción. Si la longitud del segmento es por el contrario mayor que cero, la ventana de recepción debe ser obligatoriamente mayor que cero, y en tal caso se debe cumplir que alguno de los límites del segmento se encuentre dentro de la ventana, esto es: el número de secuencia recibido es mayor o igual que el siguiente esperado y menor que la suma del

siguiente esperado mas la ventana de recepción, o que el límite superior (la suma del número de secuencia más el tamaño, menos uno) cumpla la misma desigualdad.

Si esto ocurre se trata de un segmento totalmente válido con lo que actualizamos el valor del siguiente número de secuencia esperado y devolvemos un resultado positivo.

Si el resultado de la comprobación del número de secuencia no resulta positiva, se asume que no ha llegado al otro extremo algún segmento enviado previamente, con lo que mandamos un asentimiento con los datos actualizados de la conexión, aunque esto lo veremos con detalle cuando se desarrolle el comportamiento del simulador frente a fallos.

5.5.2. TIMEOUT

Como a estas alturas es de sobra conocido, a cada octeto transmitido en un segmento de datos se le asigna un número de secuencia, y se requiere del TCP de recepción un asentimiento positivo (ack). Si dicho asentimiento no se recibe dentro de un intervalo de tiempo, los datos son retransmitidos.

Para desarrollar el código que posibilite la gestión del contador de retransmisión, nos valemos de la clase Timer del simulador de datos básico (ver punto 5.1.3.)

Para dicho contador se define tanto en el origen de la comunicación, sourceTPC, como en el destino, sinkTPC, una lista de punteros a elementos Timer, denominada timerTCP_ donde se van a almacenar un timer por cada envío de datos.

Cada vez que, bien desde sourceTCP en fillPacket o bien desde sink en recvDown, se envía al otro extremo un segmento de datos, se define un nuevo elemento auxiliar Timer (t_aux) en que almacenamos el nodo origen, el instante del evento timer, el método a ejecutar y el segmento que interviene en el método. En este caso el instante del evento será el instante del envío de datos + CONTADOR, que en el simulador se ha considerado fijo y lo suficientemente grande como para que no haya retransmisiones innecesarias. El método a ejecutar en ambos casos es retxon, método definido en ambos extremos cuya función es retransmitir en caso de que haya vencido el temporizador, lo vemos en detalle un poco más adelante. Y por último, el segmento es una copia del que se va a enviar.

Acto seguido se define una instancia de scheduler que introduzca dicho evento en la pila de eventos pendientes y se inserta t_aux en la lista timerTCP_.

Cuando vence un temporizador se ejecuta el método `retxon` de uno de los extremos. En primer lugar se debe verificar si el segmento a retransmitir es realmente un evento almacenado en la lista de retransmisión, si esto es así, se da luz verde al envío y se elimina el elemento timer de la lista. Una vez que se sabe que hay que retransmitir, lo primero es realizar una llamada al método de connection winCalculate indicándole que se ha producido una retransmisión y por tanto el nodo se encuentra en un estado de congestión, para que actúe en consecuencia reduciendo la ventana de

transmisión. El modo de actuación de winCalculate se verá en detalle en el siguiente punto. Se establece por último el origen, identificador de conexión y se le indica al otro extremo que su ventana de recepción ha sido modificada, enviando el segmento retransmitido al otro extremo.

Cada vez que llega un asentimiento a un extremo se llama al método `timeOut`, cuya misión es recorrer la lista buscando un elemento cuyo número de secuencia + longitud sea menor o igual que el número de asentimiento del segmento en recepción, en cuyo caso se desactiva dicho elemento de la lista de eventos pendientes en scheduler y se elimina de timerTCP_.

Cuando la conexión comienza el proceso de liberación de la conexión se deben borrar todos los elementos que permanezcan almacenados en la lista de eventos pendientes para que no se crucen con los segmentos de control de la liberación. Esto no debería ser necesario en una implementación real en la que cada extremo termina completamente de enviar sus datos antes de dar por finalizada la conexión. En el caso del simulador, la transferencia de datos no finaliza cuando el usuario de capa superior deja de tener datos que transmitir, sino cuando nos lo especifica el propio usuario del simulador por línea de comando, por lo que pueden quedar datos pendientes, incluso enviados sin asentir.

Para eliminar todos estos elementos “residuales”, en el instante en que cada extremo tiene constancia de que se está cerrando la conexión realiza una llamada al método timeOut especificándole como parámetro que estamos en la fase de liberación de llamada. TimeOut, recorre entonces la lista (siempre que ésta no está vacía), y elimina todos los elementos que quedan, a la vez que los desactiva para que no lleguen a producirse.

5.6. CONTROL DE FLUJO

El TCP proporciona medios de control de flujo, permitiendo al receptor tener control sobre la cantidad de datos que pueden ser enviados por el transmisor. Esto se consigue definiendo una ventana, que indica un determinado número de octetos permitidos que el transmisor puede enviar hasta que reciba asentimiento.

5.6.1. VENTANAS DE TRANSMISIÓN Y RECEPCIÓN

Tanto la ventana de transmisión como de recepción se controlan desde connection, dotando de esta forma de eficiencia al código ya que ahorramos repeticiones de grandes bloques de código en sourceTCP y sinkTCP.

Se necesita controlar la ventana de transmisión puesto que me determina en cada instante el límite de octetos que puedo transmitir sin recibir asentimiento alguno. Están definidas en connection para este fin dos variables, una que mantiene siempre el valor de la ventana inicial de transmisión indicada desde el otro extremo, y otra que va a almacenar en cada instante el valor real. Cada vez que se produce un estado de

congestión en la red, el TCP debe disminuir su ventana de transmisión, consiguiendo de este modo frenar el envío y que no se pierdan más segmentos. Todo este comportamiento, y el modo en que se actualizan estas variables de desarrollarán con todo detalle en el siguiente punto.

El control de la ventana de recepción resulta también imprescindible, puesto que me determina si el segmento recibido es, o no, válido. El valor de la ventana de recepción en cada extremo viene determinado por el usuario del simulador, pasándolo como parámetro en la línea de comandos. Sin embargo acabamos de señalar que es posible que un extremo TCP modifique su ventana de transmisión debido a un estado de congestión. Como la ventana de transmisión en un extremo no es más que la ventana de recepción en el extremo opuesto, cada vez que se modifique la ventana de transmisión habrá que avisar al otro extremo de que modifique la suya de recepción.

Cada vez que se modifica la ventana de transmisión se activa un el bit wndChange de la cabecera TCP, avisando así al otro extremo de que hemos modificado su ventana de recepción. El nuevo valor de la ventana viene almacenado en el campo newWnd del segmento TCP.

Como ya se ha comentado en numerosas ocasiones, cada vez que llega un segmento al recvDown de uno de los extremos, tras comprobar que no contiene errores, se verifica si ha habido cambios en la ventana de recepción comprobando el valor del bit wndChange. Si está activo, se actualiza el tamaño de la ventana de recepción.

5.6.2. CONTROL DE LA CONGESTIÓN

El estándar TCP no especifica ningún mecanismo de control de flujo específico, sólo que en caso de que haya mucho tráfico se debe bajar el tamaño de la ventana, puesto que una ventana grande anima a las transmisiones.

Existe un mecanismo ampliamente utilizado para controlar la congestión en TCP durante el intercambio de datos, denominado AIMD. Dicho mecanismo se basa en que rara vez los segmentos se pierden debido a errores en la transmisión, por tanto una retransmisión debido al vencimiento de un contador de retransmisión significa que la red está congestionada.

El modo de operación consiste en que cuando ocurre un vencimiento de temporizador se divide el tamaño de la ventana en dos (decremento multiplicativo) alejándonos así rápidamente de situaciones de saturación. A partir de entonces entramos en un estado de recuperación de la ventana, incrementándola un poco cada vez que llega un asentimiento (incremento aditivo). Este incremento se calcula como el tamaño máximo de segmento al cuadrado, dividido entre la ventana de congestión.

El simulador controla todos estos cambios mediante el método `winCalculate` de connection. Como hemos visto, este método comprueba si tenemos por primera vez la conexión establecida, en estado de congestión o de recuperación, actuando en consecuencia.

El parámetro congestion que pasamos a winCalculate lo activa source o sink en caso de entrar en el método retxon, esto es, en caso de que haya ocurrido el vencimiento de un temporizador y se haya producido en consecuencia una retransmisión.

Si congestion es true, en winCalculate, ponemos la variable recuperacion_ de connection a true en primer lugar. Actualizamos entonces la ventana real de transmisión como la mitad del valor que tenía. Comprobamos en este punto si con este decremento la ventana ha llegado a tomar valor cero, en cuyo caso fijamos el tamaño de segmento también a cero. Si la ventana es mayor que cero, comprobamos si el tamaño actual del segmento es mayor que el nuevo tamaño de la ventana de transmisión, fijando en este caso el tamaño del segmento al mismo valor que la ventana. Como entramos en este método en el caso de congestión desde el método retxon de uno de los extremos, se activa en el segmento a retransmitir el bit de cabecera wndChange y se introduce el nuevo valor para que actualice el otro extremo su ventana de recepción.

Entramos en el método en estado de recuperación de ventana siempre que tengamos una ventana de transmisión menor que la indicada inicialmente desde el otro extremo a raíz de un estado de congestión. En este estado, se calcula en primer lugar el número de octetos que vamos a incrementar la ventana, que es el resultado de la división entera entre el tamaño de segmento al cuadrado y la ventana real de transmisión; añadiendo a continuación este incremento a la ventana real de transmisión. Comprobamos si la nueva ventana de transmisión es mayor o igual que la inicial, finalizando en tal caso el estado de recuperación. Fijamos en este caso el valor real de la ventana de transmisión igual al tamaño inicial y el tamaño del segmento también será el mismo que si no hubiera pasado ningún estado de congestión. Por último devolvemos como resultado de la llamada al método true, para que los nodos TCP tengan constancia de que se ha modificado el valor de la ventana de transmisión y deben enviarla al otro extremo.

En sinkTCP la idea es bastante sencilla, puesto que la recepción está íntimamente ligada con la transmisión. Cada vez que se recibe un segmento, se envía otro como respuesta. Como se realiza una llamada a winCalculate cada vez que se recibe un segmento, basta comprobar el resultado de esta llamada a la hora de establecer los valores del nuevo segmento para ver si tenemos que activar el bit wndChange y enviar en newWnd el nuevo valor de la ventana.

En sourceTCP la idea se complica ligeramente al estar completamente independientes la recepción y la transmisión. El cambio es que sourceTCP define un flag como variable de instancia que va a controlar si hay que enviar en el segmento de datos un nuevo valor de ventana. Este flag almacena el resultado de la llamada a winCalculate, y se comprueba su valor siempre que se llenan los campos del segmento en fillPacket de sourceTCP. En caso de que sea afirmativo, al igual que en el caso anterior, se activa el bit wndChange y se envía en newWnd el nuevo valor de la ventana.

5.7. SITUACIONES DE ERROR

Para finalizar vamos a comprobar cómo se comporta el simulador frente a las distintas situaciones de error que nos podemos encontrar a lo largo de la simulación.

Estaremos frente a una situación de error siempre que nos llegue un segmento que no tiene nada que ver con la conexión. El propio estándar nos define cuales son estos segmentos y cómo actuar cuando llega uno de ellos (ver punto 2.2.5. de la memoria).

Se especifica que si la conexión se encuentra en un estado no sincronizado y recibe un segmento con un ack inaceptable se envía un mensaje especial reset, permaneciendo la conexión en el mismo estado. El simulador lo que hace es que si se produce un fallo en la comprobación del segmento recibido estando la conexión en estado listen, syn-sent o syn-received, informa de la situación al usuario mediante un mensaje por pantalla y se activa el flag reset.

De igual forma se especifica que si la conexión se encuentra en un estado sincronizado y produce error la validación del segmento recibido sacará un mensaje de asentimiento con el número de secuencia de envío actual y un número de asentimiento indicando el siguiente número de secuencia esperado para recibir, permaneciendo la conexión en el mismo estado. El simulador comprueba si se produce un error en la comprobación del segmento recibido durante un estado establish, fin-wait-1, fin-wait-2, close-wait, closing, last-ack o time-wait, en cuyo caso informa del error por pantalla al usuario del simulador y activa el flag asentimiento.

En ambos casos una llamada al método PackProcessing daría como resultado un segmento nulo, con lo que se realiza una llamada al método returnError de connection que devuelve true siempre que el flag reset o asentimiento estén activos. Cuando esto ocurre, se llama al método errorMsg también de connection al que se le pasa como parámetro el segmento inicial recibido, que formará el mensaje de error según este sea un reset o un asentimiento.

Si es reset el flag activo, se crea un mensaje con todos los bits de cabecera a false mediante ControlPacket, se activa el bit rst de cabecera y se establece el número de secuencia del paquete, que será el número de asentimiento recibido. Por último se desactiva la bandera reset.

Si el flag activo es el de asentimiento, se crea un mensaje de control activando simplemente el bit ack de cabecera mediante ControlPacket, se fija el número de secuencia del paquete igual al número de secuencia recibido, y el de asentimiento con el siguiente esperado actual, se actualiza el siguiente número de secuencia a enviar y se desactiva la bandera de asentimiento.

De nuevo en recvDown, se fija el origen e identificador de conexión y se envía al otro extremo.

Cada vez que se recibe un segmento en PackProcessing se comprueba en primer lugar si tiene errores o se trata de un segmento de error para actuar en consecuencia. Si presenta error de cabecera (error de checksum) se ignora el segmento y finaliza el método. En caso contrario se comprueba si el segmento tiene el bit rst activo, llamándose en ese caso al método reset de connection que actúa según el estado de conexión en que se encuentre: si es syn-sent o syn-received, provoca sólo que el estado vuelva a Listen; mientras que si se encuentra en uno de los estados sincronizados aborta la conexión (activa el flag abort) e informa al usuario. A continuación se comprueba en recvDown si se indica que se aborte la conexión, y en este punto difieren ambos extremos:
-sourceTCP aborta la conexión parando el envío de datos.
-sinkTCP sin embargo no tiene esa capacidad, ya que envía siempre que recibe un segmento válido. El único modo que tiene el destino de datos de abortar la conexión es solicitándole a sourceTCP que lo haga. Haciendo uso del mismo método errorMsg, indicándole que se trata de un mensaje de abort, se construye un segmento con todos los bits de control a false mediante ControlPacket, activando luego el bit abort. Se establece el origen, la dirección y el identificador de conexión y se envía a sourceTCP, donde se comprueba si este bit está activo antes que cualquier otra cosa, parando inmediatamente el envío de datos.