ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

IMPLEMENTACIÓN DE PROTOCOLOS DE LA CAPA DE ENLACE DE DATOS EN LOS SIMULADORES OMNET++ Y NS-2

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y REDES DE INFORMACIÓN

JOHANNA PATRICIA FONTE AREQUIPA

johafont@gmail.com

FABIOLA ELIZABETH MORA MULLA

fabiola.mora.msn@gmail.com

DIRECTOR: IVÁN BERNAL CARRILLO, Ph.D
ibernal@fie.epn.edu.ec

QUITO, JUNIO 2008

DECLARACIÓN

Nosotras, Johanna Patricia Fonte Arequipa y Fabiola Elizabeth Mora Mulla, declaramos bajo juramento que el trabajo aquí descrito es de nuestra autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que hemos consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedemos nuestros derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

Johanna Patrícia Fonte Arequipa

Fabiola Elizabeth Mora Mulla

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Johanna Patricia Fonte Arequipa y Fabiola Elizabeth Mora Mulla, bajo mi supervisión.

Iván Bernal Carrillo, Ph.D.
DIRECTOR DE PROYECTO

AGRADECIMIENTO

En primer lugar, agradezco a mi familia que ha estado siempre a mi lado, en los buenos y malos momentos.

Agradezco con especial cariño a todos los miembros de la familia Chamorro quienes han sido un constante apoyo en toda mi trayectoria de estudios.

De igual manera, agradezco la paciencia y perseverancia de Johanna y su familia, quien además ha demostrado ser más que una amiga.

No puedo dejar pasar la oportunidad para agradecer a mi profesor, Iván Bernal Ph.D por el tiempo, paciencia y apoyo que nos ha dado durante la realización de este proyecto.

Fabiola Mora

Agradezco a Dios por la vida y por guiar mi camino, a Iván Bernal Ph.D por el incentivo y esfuerzo en dirigir el presente proyecto de titulación, a Fabiola Mora por la dedicación y esfuerzo.

Johanna Fonte

DEDICATORIA

Todo esfuerzo lo dedico en primer lugar a Dios.

Dedico también este esfuerzo a la Dra. Genoveva Arias de Chamorro, Clemente Chamorro, mi hermano Darwin, pero especialmente a la mujer que ha sido en mi vida la fuente de inspiración para alcanzar las metas que me he trazado, mi madre querida, Lida.

Fabiola Mora

A mis padres por su amor, apoyo y comprensión, a mi hermano y hermana por la motivación y confianza, a mis sobrinos por la alegría que brindan a mi familia.

Johanna Fonte

CONTENIDO

ÍNDICE DE FIGURAS

CAPÍTULO 1

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DE CÓDIGOS

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DE EJEMPLOS

CAPÍTULO 1

ÍNDICE DEL CÓDIGO DEL PROTOCOLO “SIMPLEX SIN RESTRICCIONES”

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DEL CÓDIGO DEL PROTOCOLO “SIMPLEX DE PARADA Y ESPERA”

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DEL CÓDIGO DEL PROTOCOLO “SIMPLEX DE PARADA Y ESPERA PARA UN CANAL CON RUIDO”

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DEL CÓDIGO DEL PROTOCOLO DE “VENTANA CORREDIZA DE UN BIT”

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DEL CÓDIGO DEL PROTOCOLO DE “VENTANA CORREDIZA CON RETROCESO N”

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DEL CÓDIGO DEL PROTOCOLO DE “VENTANA CORREDIZA DE REPETICIÓN SELECTIVA”

CAPÍTULO 2

CAPÍTULO 3

ÍNDICE DE LA CONFIGURACIÓN DE LOS ESCENARIOS DE SIMULACIÓN

CAPÍTULO 2

CAPÍTULO 3

RESUMEN

En la actualidad los simuladores juegan un papel fundamental ya que permiten realizar pruebas a distintos niveles en modelos simplificados de la realidad, reduciendo así los tiempos de desarrollo y aumentando la fiabilidad de los resultados. Existen diversos tipos de simuladores, específicamente en el área de las redes de telecomunicaciones y de datos, en los cuales se puede modelar cualquier elemento de un sistema dado y a través de la simulación obtener resultados útiles para su análisis.

En particular, existen simuladores de distribución gratuita que son muy utilizados en el área académica ya que permiten que el usuario pueda contribuir con sus propias implementaciones. Entre los simuladores más utilizados para redes de telecomunicaciones y datos están OMNET++ y NS-2

Con este trabajo se busca facilitar un primer contacto con simuladores de tipo académico relacionados con redes, específicamente con los simuladores OMNET++ y NS-2. Esto incluye la comprensión de la importancia de los simuladores, así como también el incentivar al usuario el uso de estas herramientas ya sea utilizando los componentes que vienen originalmente con los simuladores o desarrollando nuevos componentes, los cuales permitirán al usuario plantear su propio escenario de simulación.

En este trabajo se implementan los protocolos de capa de enlace de datos presentados en el libro de “Redes de Computadoras” del autor Andrew Tanembaum en los simuladores OMNET++ y NS-2. Las implementaciones se las realiza de acuerdo a las facilidades que ofrecen cada uno de los simuladores en estudio, además se presenta el procedimiento para la configuración de los escenarios en los que se simula los protocolos propuestos y se verifica que los resultados obtenidos estén acorde a lo esperado.

PRESENTACIÓN

En este proyecto se presenta la metodología para implementar protocolos de capa de enlace de datos en simuladores de redes. En particular, el trabajo se centra en introducir nuevos componentes que representen a protocolos en los simuladores OMNET++ y NS-2.

El proyecto se compone de cuatro capítulos. El primer capítulo presenta el escenario y la motivación en el que se enmarca este trabajo, dando una introducción a cada uno de los simuladores, los lenguajes de programación que utilizan, las herramientas con que cuenta cada uno de ellos para la presentación de los resultados y las principales clases dada la naturaleza orientada a objetos de OMNET++ y NS-2 de las que dependen las nuevas implementaciones en los respectivos simuladores.

Los protocolos de la capa de enlace de datos a implementarse son seis, los cuales han sido clasificados en dos grupos: tres protocolos elementales de enlace de datos y tres protocolos de ventana corrediza.

Tanto en el Capítulo 2 como en el Capítulo 3 se describen los procedimientos para implementar estos protocolos de la capa de enlace de datos.

En el Capítulo 2 se presenta la implementación misma de cada uno de los protocolos en el simulador OMNET++, para ello se realiza una breve introducción de los elementos que intervienen en el escenario de simulación de los protocolos planteados, logrando determinar que es lo que se debe agregar a la funcionalidad básica que provee el simulador. La funcionalidad de los protocolos se implementa en lenguaje de programación C++, la configuración del escenario de simulación se realiza mediante el editor gráfico de lenguaje NED (GNED), además, se describe la sintaxis que se debe seguir para la configuración de los parámetros de los protocolos para la simulación, se describen los resultados obtenidos de la simulación comprobando que estos sean coherentes con la naturaleza del protocolo. Finalmente, se describe la forma de interacción de un nuevo componente (protocolo) con el simulador, la cual se complementa con un diagrama de secuencia que resume el proceso.

En el Capítulo 3 se realiza una breve introducción de las clases que implementan los principales elementos que intervienen en un escenario de simulación, en base a esto se determina de donde partir para la implementación de los nuevos protocolos planteados. La funcionalidad de los protocolos se implementa en lenguaje de programación C++, la configuración del escenario de simulación y de los parámetros de los protocolos se realiza en un script para lo cual se hace uso del lenguaje de programación OTcl; se describen los resultados obtenidos de la simulación comprobando que estos sean coherentes con la naturaleza del protocolo y, finalmente, se describe la forma de interacción de un nuevo protocolo con el simulador, la cual se complementa con un diagrama de secuencia que resume el proceso.

Cabe mencionar que los protocolos que se implementan se presentan de forma secuencial de acuerdo a un grado de complejidad creciente, lo que requiere el uso de nuevas funciones y variables; por tanto, se considera que en cada protocolo se explicará únicamente lo que se agregue a la implementación respecto al protocolo anterior.

Finalmente, en el Capítulo 4 se enumeran las conclusiones obtenidas respecto del trabajo realizado y se presentan algunas recomendaciones que pueden ser útiles para futuras implementaciones.

También se incluyen anexos que contienen: el procedimiento de instalación de los simuladores en estudio, copias del Capítulo 3 del libro “Redes de Computadoras” del autor Andrew Tanembaum y la descripción de los objetos de red básicos en NS-2.

Adicionalmente, se adjunta un CD el cual contiene un tutorial en formato PDF indexado, todo lo realizado en este trabajo y que puede ser de ayuda a quien desee solo simular o implementar nuevos componentes en los simuladores.

CAPÍTULO 1

CARACTERÍSTICAS GENERALES DE LOS SIMULADORES OMNET++ Y NS-2

1.1. INTRODUCCIÓN

Las herramientas de simulación facilitan el aprendizaje de las redes de telecomunicaciones y datos, permitiendo que el alumno asimile diversos conceptos, muchas veces un tanto difíciles de entender, como el funcionamiento de los protocolos para las diferentes capas de los modelos de referencia para redes.

Sin lugar a duda, las herramientas de simulación proporcionan un marco de trabajo en el que el alumno es capaz de visualizar los conceptos, a la vez que analiza las características principales de las redes.

Entre los simuladores de mayor difusión se encuentran: OMNET++ y NS-2, ya que al ser de distribución gratuita, son accesibles a los interesados; además, presentan una implementación robusta y extensible que sirve de base para futuras implementaciones **[1, 2]**.

1.2. SIMULADOR OMNET++

OMNET++ (*Objective Modular Network Testbed in C++*) es un sistema que permite modelar y simular eventos discretos en redes de datos, orientado a objetos (basado en el lenguaje C++). Se basa en el uso de módulos[1], creados mediante el lenguaje C++, y que luego son relacionados mediante el lenguaje NED[2]. Los módulos pueden ser simples o compuestos (éstos serán descritos posteriormente).

Un modelo en OMNET++ consiste de: la descripción de la topología física, la creación de módulos jerárquicos y la definición de los mensajes a enviarse entre éstos.

Este simulador permite:

- Modelar el tráfico de información a ofrecerse a las redes a simular.
- Modelar los protocolos de redes de telecomunicaciones y de datos.
- Modelar multiprocesadores y otros sistemas de hardware distribuido.
- Modelar redes de colas.

Aunque permite modelar cualquier sistema, está especialmente diseñado para el modelado y simulación de protocolos de redes.

Este simulador provee una librería de clases que representa al *kernel* de simulación y la interfaz de usuario.

1.2.1. JERARQUÍA DE MÓDULOS

Un modelo en OMNET++ consiste de módulos enlazados jerárquicamente y que se comunican entre sí. Los modelos a menudo son referidos como *networks.* Existen tres tipos de módulos:

1. **Módulo del sistema:** Es el módulo de nivel superior; contiene a los módulos compuestos y simples.

[1] Componente o conjunto de componentes que se combinan para conformar un modelo a simular.
[2] NED: *NEtwork Description.*

2. **Módulo compuesto:** Los módulos compuestos son agrupaciones de dos o más módulos simples y/o compuestos, permitiendo obtener un sistema de niveles jerárquico.
3. **Módulo simple:** Los módulos simples son los elementos activos[1] que se utilizan para construir un módulo compuesto (por ejemplo la implementación de un protocolo).

En la Figura 1.1 se presenta la jerarquía de módulos en OMNET++.

Figura 1.1 Jerarquía de módulos para un modelo en OMNET++.

1.2.2. LENGUAJES

Para la creación de un modelo a simular mediante OMNET++, se hace uso de los lenguajes C++ y NED.

1.2.2.1. Lenguaje C++

Se utiliza el lenguaje C++ para implementar el comportamiento de los módulos que forman parte del modelo a simular, específicamente de los módulos simples.

No se detalla la sintaxis del lenguaje, y se sobreentiende que el lector debe tener un conocimiento básico de C++.

1.2.2.2. Lenguaje NED

Este lenguaje se utiliza para modelar y describir, la estructura de las redes de telecomunicaciones y de datos.

Este lenguaje facilita la descripción modular de una red. Entre los componentes de la descripción modular de una red están:

- Módulos simples y compuestos (que pueden ser reutilizados en la descripción de otra red).

[1] Elementos en los que se implementa el comportamiento de un objeto de red.

- Las conexiones entre los módulos.

Los archivos que contienen la descripción de la red, tienen una extensión .ned y son traducidos a lenguaje C++ por medio del compilador *nedtool*, de esta manera podrán ser enlazados dentro del archivo ejecutable de simulación.

1.2.2.2.1. Sintaxis

El modelo a ser descrito con el lenguaje NED se basa en la definición de las directivas de importación, la definición de módulos simples, la definición de módulos compuestos y la definición de la red.

Directivas de importación: Se utilizan para importar declaraciones de otros archivos que contienen la descripción de otras redes. Después de importar la descripción de una red, los componentes como los canales y los módulos que hacen parte de dicho archivo, pueden ser utilizados. El nombre de los archivos puede especificarse sin la extensión .ned.

La sintaxis para importar un archivo es la siguiente:

```
import "ethernet"; // Importa el archivo ethernet.ned
```

Definición de módulos simples: La definición de un módulo simple empieza con la palabra reservada *simple*, seguido del nombre del mismo (por convención empiezan con una letra mayúscula), y se finaliza con la palabra *endsimple* (ver el Ejemplo 1.1).

Además, para un módulo se pueden especificar sus parámetros y sus compuertas[1]. Cabe indicar que la implementación de la funcionalidad de un módulo simple se lo hace mediante el lenguaje C++ y utilizando la librería omnetpp.h[2].

Los parámetros pueden ser utilizados por el o los algoritmos del módulo; y son identificados por nombres que por convención empiezan con letras minúsculas.

Para especificar una sección de parámetros se utiliza la palabra reservada *parameters* como se presenta en el Ejemplo 1.1.

---

[1] Representa el punto de conexión entre módulos.
[2] Librería del *kernel* de simulación.

Un parámetro puede ser de tipo: *numeric, numeric const* (o simplemente *const*), *bool, string o xml*. Si el tipo es omitido se asume que es *numeric* **[1]**.

La configuración de los parámetros (asignación de valores iniciales) se la puede realizar directamente en el archivo .ned, en el archivo omnetpp.ini[1], si los valores no son asignados en los archivos antes mencionados, éstos deberán ser asignados en el cuadro de diálogo que aparece en tiempo de ejecución en el interfaz gráfico *Tkenv* (ver Figura 1.2).

Figura 1.2 Cuadro de diálogo para asignar el valor al parámetro *timer* en la interfaz *Tkenv*.

La sección de compuertas se especifica mediante la palabra reservada *gates* (ver el Ejemplo 1.1). Las compuertas pueden ser de entrada o salida. En su declaración se debe anteceder la palabra *in* u *out* al nombre que se le asigna a la compuerta.

En el Ejemplo 1.1 se presenta la definición de un módulo simple, incluyendo la declaración de sus parámetros y sus compuertas.

```
// Se crea un módulo simple de nombre Protocolo
simple Protocolo
// Se define la sección en la que se especifican los parámetros
parameters:
    timer=0.015,
    nombre: string;
// Se define la sección en la que se especifican las compuertas
gates:
    out: salida; //Compuerta de salida
    in: entrada; //Compuerta de entrada
endsimple
```

Ejemplo 1.1 Definición de un módulo simple en lenguaje NED.

Definición de módulos compuestos: La definición de un módulo compuesto empieza con la palabra reservada *module*, seguida del nombre del mismo (por convención empiezan con letra mayúscula), y se finaliza con la palabra *endmodule* (ver el Ejemplo 1.2).

[1] Archivo que contiene los valores iniciales a asignarse a los parámetros de los módulos creados en lenguaje NED.

Para un módulo compuesto se pueden especificar: parámetros, compuertas, submódulos y conexiones.

El concepto y definición de los parámetros y compuertas en los módulos compuestos es similar al mencionado en los módulos simples y son utilizados solo en caso de ser necesario. Típicamente, los valores de los parámetros de los módulos compuestos son pasados a los parámetros de los submódulos[1].

Para especificar una sección de submódulos se utiliza la palabra reservada *submodules.*

Las conexiones entre los módulos se definen en la sección *connections.* Una conexión enlaza las compuertas de entrada y salida de los módulos que se desea conectar. Para cada conexión se puede configurar los siguientes atributos: *delay* (retardo en segundos), *error* (probabilidad de que un bit sea transmitido incorrectamente) y *datarate* (velocidad de transmisión en bit/segundo).

La sintaxis para definir un módulo compuesto se indica en el Ejemplo 1.2.

```
// Definición de un módulo de nombre Prot
module Prot
      parameters:
      timer;
// Sección de submódulos
       submodules:
// Instancia de un módulo simple(Protocolo) de nombre A
        A: Protocolo;
      parameters:
              timer = timer;
// Instancia de un módulo simple(Protocolo) de nombre B
        B: Protocolo;
              timer = timer;
// Sección de conexiones
    connections:
// Se configura un canal de comunicación entre A y B, para el cual se
// configura el valor del tiempo de propagación
        A.salida --> delay 0.50 --> B.entrada;

// Se configura un canal de comunicación entre B y A, para el cual se
// configura el valor del tiempo de propagación, se configura la
// probabilidad de error de un bit transmitido en este canal y se
// configura la velocidad de transmisión
        B.salida --> delay 0.50 error 0.01 datarate 200--> A.entrada;
endmodule
```

Ejemplo 1.2 Definición de un módulo compuesto en lenguaje NED.

[1] Los submódulos consisten en instancias de cualquier tipo de módulo que puede ser simple o compuesto.

Definición de la red: Es una instancia de un tipo de módulo previamente definido y representa al módulo del sistema.

Su declaración es necesaria para lograr la simulación de un modelo diseñado con el lenguaje NED.

La definición de una red empieza con la palabra reservada *network*, seguida del nombre del mismo (por convención empiezan con letra mayúscula), y se finaliza con la palabra *endnetwork* (ver el Ejemplo 1.3).

```
// Definición de una red de nombre Network, instancia del módulo
// compuesto Prot

network Network : Prot
endnetwork
```

Ejemplo 1.3 Definición de una red en lenguaje NED.

Adicionalmente, el lenguaje NED tiene un editor gráfico asociado llamado GNED[1]; este editor facilita al usuario el diseño de la topología de la red y trabaja con una estructura interna de datos que contiene el lenguaje NED.

Una vez que se guardan los archivos con la descripción gráfica de la red, este editor genera un archivo que contiene la traducción de la topología en términos del lenguaje NED, para que pueda ser utilizado en las demás aplicaciones.

La aplicación GNED cuenta con una interfaz gráfica, compuesta por una barra de herramientas, un área de trabajo, un listado de archivos generados y una ventana que permite visualizar el código NED del modelo que se está construyendo de manera gráfica; en la Figura 1.3 y en la Figura 1.4 se presenta lo antes descrito.

[1] GNED: *Graphical NED Editor*.

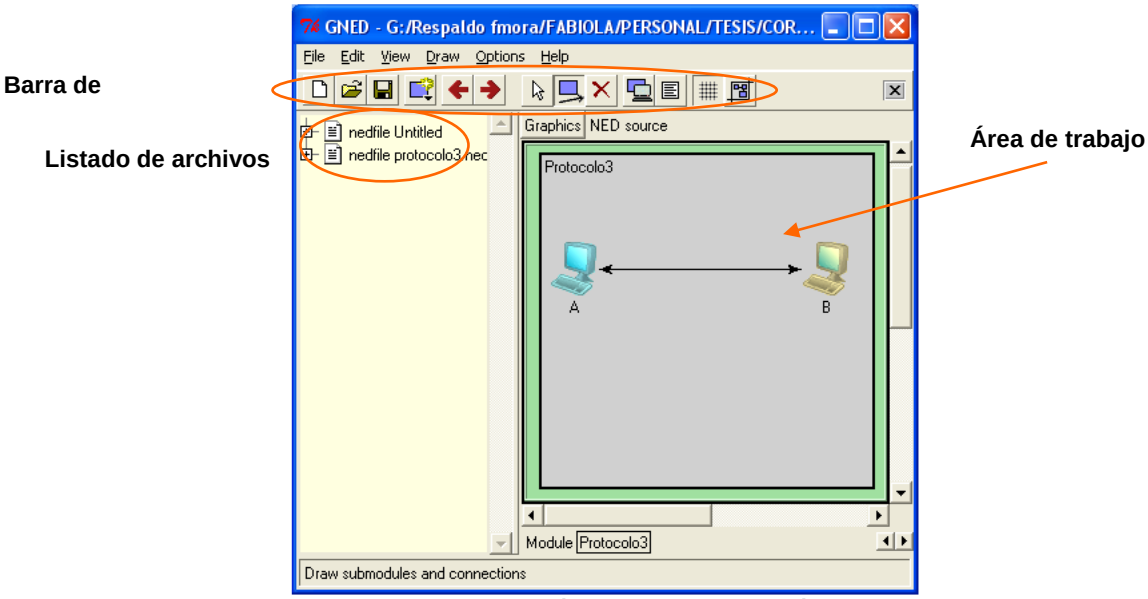


Figura 1.3 Interfaz gráfica de la aplicación GNED.

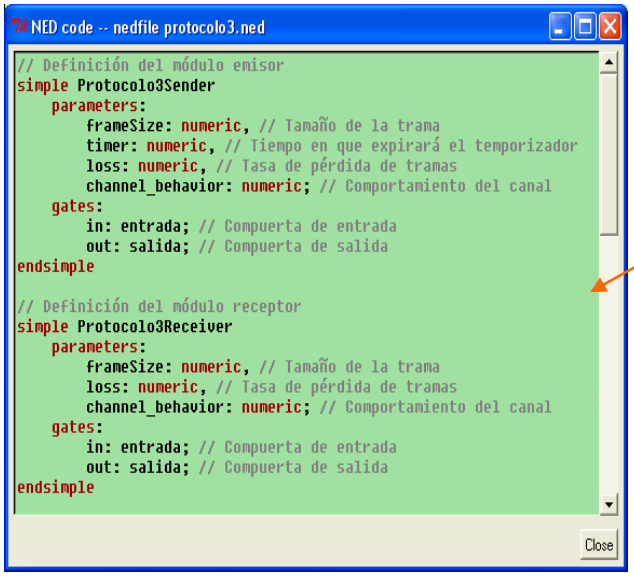


Figura 1.4 Visualización de la descripción en lenguaje NED del modelo construido gráficamente.

1.2.3. CLASES PRINCIPALES

1.2.3.1. cSimpleModule

Es una de las clases que conforman el *kernel* de simulación, es decir forma parte de la librería de clases del simulador OMNET++. Esta clase define funciones que deberán ser redefinidas por las clases que se deriven de ésta. Básicamente, una clase derivada implementará el comportamiento del módulo simple que forma parte de la descripción de la red.

Las funciones definidas en la clase *cSimpleModule* tienen como objetivo la generación de eventos y la reacción ante ellos.

Las funciones virtuales a ser redefinidas son:

- **initialize():** En esta función se hará la inicialización de las variables y parámetros de los módulos.
- **handleMessage():** En esta función se hará el tratamiento de los eventos que lleguen al módulo. Es una de las funciones más importantes ya que en ella se concentra la mayor parte de la funcionalidad del módulo.
- **finish():** La implementación de esta función es opcional y debe ser invocada cuando las simulaciones finalizan satisfactoriamente. Es utilizada para la recopilación de resultados escalares en un archivo.

La funcionalidad de esta clase se profundizará en el Capítulo 2.

1.2.3.2. cMessage

Los objetos de esta clase pueden ser utilizados para representar unidades de datos como: mensajes, paquetes, tramas, etc; los cuales pueden ser enviados de un módulo a otro o al mismo módulo.

La funcionalidad que esta clase proporciona será explicada en el Capítulo 2.

1.2.3.3. cMessageHeap

La finalidad de esta clase es servir como una estructura de datos en la que se almacenan los eventos planificados (por ejemplo el tiempo planificado para la llegada de una unidad de datos), representa el FES (*Future Event Set*) o FEL (*Future Event List*) **[1]**.

En la Figura 1.5 se presenta el diagrama de flujo para el consumo de eventos que se encuentren almacenados en la estructura FEL.

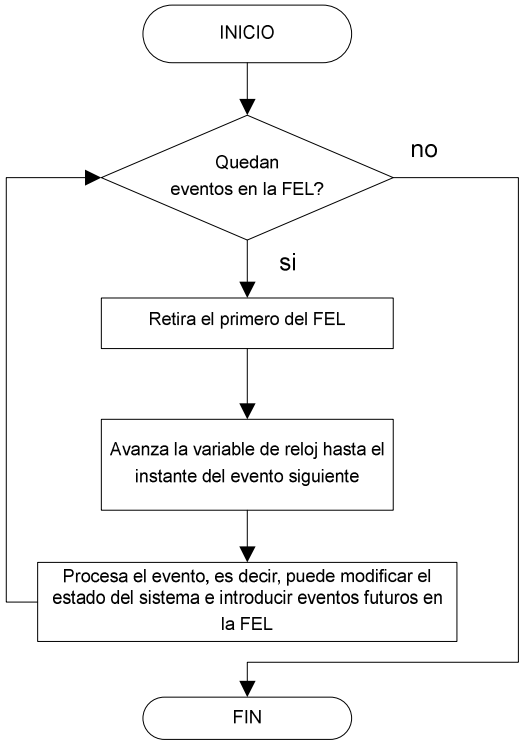


Figura 1.5 Consumo de eventos de la estructura FEL.

1.2.4. INTERFAZ DE USUARIO

Las simulaciones de OMNET++ pueden ser ejecutadas desde dos interfaces de usuario diferentes:

***Tkenv*:** Es una interfaz gráfica de usuario que proporciona ejecuciones interactivas de simulación, obtención de trazas y depuración. Está basada en Tcl/Tk[1] **[3]**.

***Cmdenv*:** Es una interfaz de usuario que se basa en la ejecución por lotes mediante línea de comandos.

A continuación se describirá sólo la interfaz de usuario *Tkenv* debido a que es amigable para el usuario y facilita la manipulación de la ejecución de la simulación.

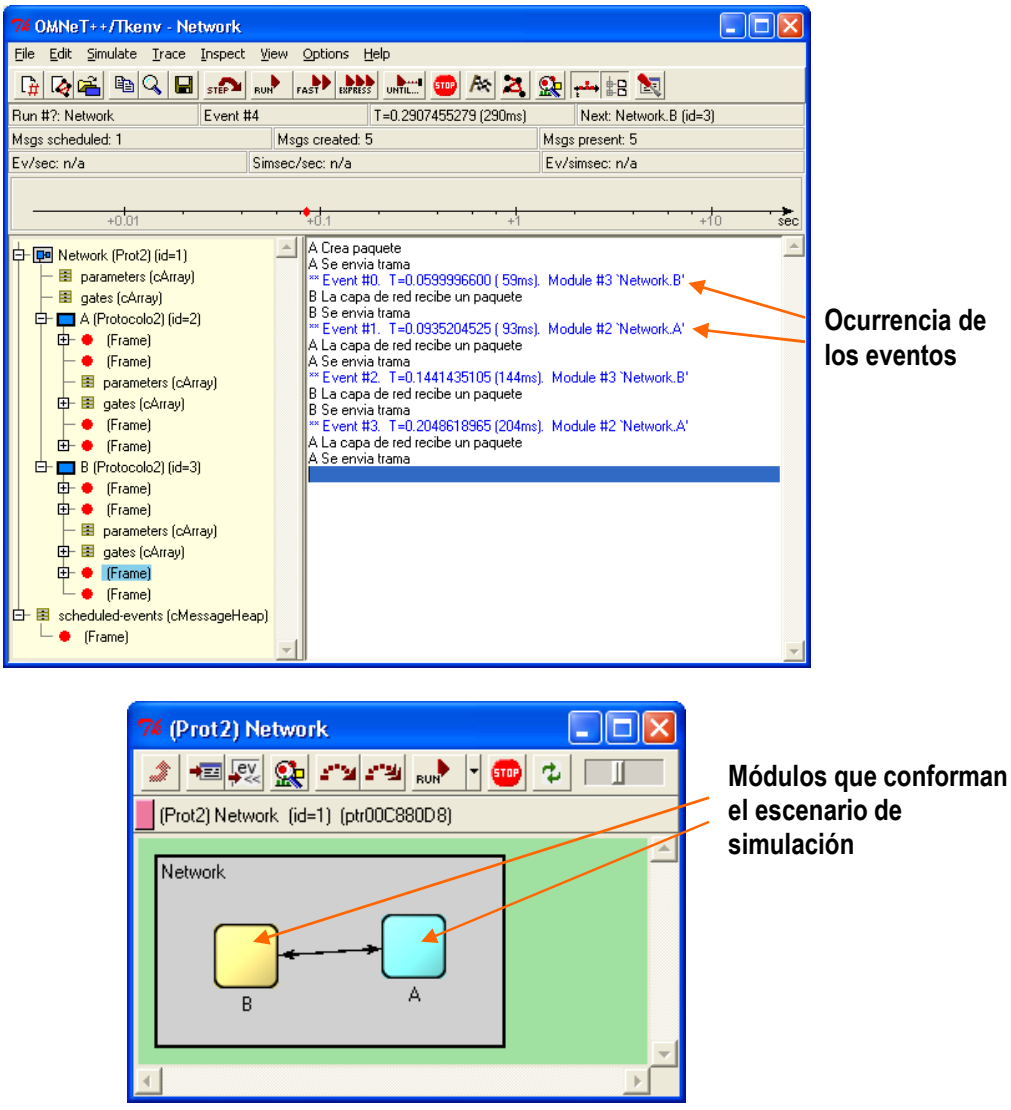
Características importantes de la interfaz *Tkenv* son:

- Permite observar los mensajes de depuración de cada módulo a medida que progresa el tiempo de la simulación.

[1] **Tcl/Tk : Lenguajes interpretados de programación visual, que genera código portable. Es desarrollado por la empresa *Sun Microsystem*** [4]**.**

- Permite la ejecución de evento por evento.
- Provee una ventana que permite examinar y alterar objetos y variables en el modelo.
- Provee una barra de herramientas para iniciar, detener, y pausar la animación.
- Para un instante de tiempo seleccionado, permite observar los detalles de la ejecución de la simulación (objetos, variables, etc)

A continuación, en la Figura 1.6 se presenta un ejemplo de simulación en la interfaz *Tkenv*.



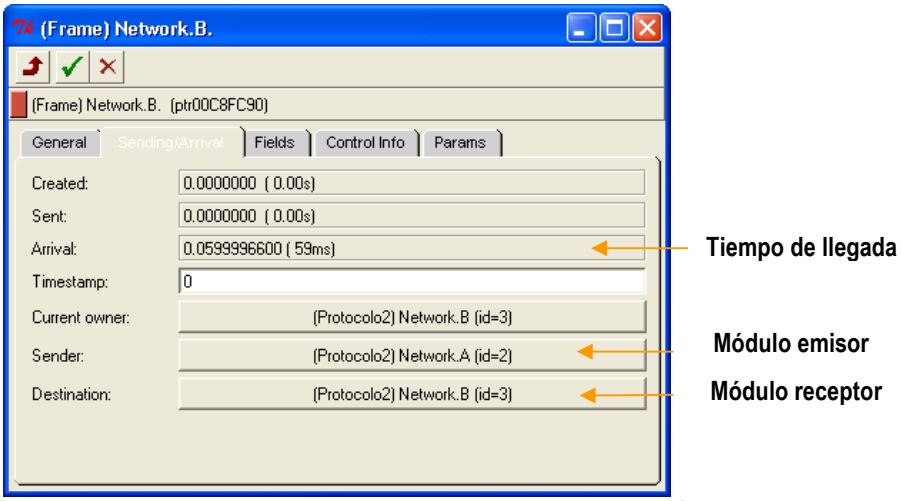


Figura 1.6 Pantallas que se obtienen al simular en la interfaz gráfica *Tkenv*.

Las opciones más importantes de su barra de herramientas son:

- **Step:** Ejecuta el siguiente evento de la simulación; como por ejemplo; generar un mensaje o enviar una trama.
- **Run**: Ejecuta la simulación a velocidad normal.
- **Fast:** Ejecuta la simulación a la máxima velocidad posible, mostrando los mensajes de depuración del protocolo en la pantalla principal.
- **Express:** Ejecuta la simulación a la máxima velocidad posible, pero no se muestran los mensajes que se obtienen de la simulación, sólo los resultados finales de la simulación.
- **Until:** Ejecuta la simulación hasta un instante determinado.
- **Stop:** Detiene la simulación.

Los resultados de la simulación se pueden almacenar en archivos con extensión .sca o .vec según sea el propósito del análisis; para visualizar los archivos antes mencionados se utiliza las aplicaciones *Scalars* y *Plove* respectivamente, provistas por OMNET++.

1.3. SIMULADOR NS-2

NS-2 (*Network Simulator Version* 2) es un simulador de eventos discretos[1] orientado a objetos, escrito en C++ con un intérprete OTcl[2], que representa la interfaz hacia el usuario.

NS-2 es una de las herramientas más utilizadas en el ámbito académico y de investigación, ya que permite simular:

- Un amplio número de protocolos para redes de telecomunicaciones y de datos.
- Tipos de redes (inalámbricas, cableadas, satelitales).
- Elementos de red (enlaces, nodos).
- Fuentes de tráfico (telnet, web, CBR[3], ó VBR[4]) **[5]**.
- Además, por ser de código abierto, facilita a los usuarios extender su funcionalidad y/o crear nuevos objetos de red[5].

1.3.1. LENGUAJES

NS-2 se basa en dos lenguajes de programación: lenguaje C++ y lenguaje OTcl.

1.3.1.1. Lenguaje C++

NS-2 está implementado en el lenguaje de programación C++; además, a través de éste los usuarios de NS-2 pueden crear nuevas clases, en las cuales se implementa la funcionalidad para los diferentes objetos de red. Una vez compiladas las clases, estarán disponibles para el intérprete OTcl a través de un mecanismo para vincular objetos C++ y OTcl (*linkage*), el cual será explicado en el Capítulo 3.

La utilización de C++ se debe a que al ser un lenguaje compilado es más eficiente en tiempo de ejecución, aunque desde el punto de vista del usuario puede ser

---

[1] Un suceso que se genera en un instante de tiempo particular.
[2] OTcl: versión orientada a objetos de Tcl (*Tool Command Language*).
[3] *Constant Bit Rate.*
[4] *Variable Bit Rate.*
[5] Se consideran objetos de red a los protocolos, tipos de redes, elementos de red y modelos de tráfico.

más lento para adaptarse a los requerimientos cambiantes (configuración de los parámetros).

En este trabajo no se detalla la sintaxis del lenguaje, y se sobreentiende que el lector tiene un conocimiento básico del mismo.

1.3.1.2. Lenguaje OTcl

NS-2 utiliza el lenguaje de programación OTcl para desarrollar los *scripts*[1] de simulación, en los que se configuran las características de los escenarios (topología, parámetros de los enlaces, protocolos, modelos de tráfico), así como también para establecer la planificación de los eventos.

Su utilización se debe a que, al ser un lenguaje interpretado[2], permite realizar cambios de manera rápida e interactiva, sin embargo, es más lento en tiempo de ejecución.

1.3.1.2.1. Sintaxis

A continuación se presenta una guía básica para la programación en lenguaje OTcl. Para mayores detalles puede referirse al manual sugerido en **[5]**.

La sintaxis básica para un comando de OTcl es:

```
comando arg1 arg2 arg3…
```

Definición de variables y asignación de valores

En el Ejemplo 1.4 se presenta un *script* OTcl, en el que se realiza la definición de variables y asignación de valores.

```
# Se asigna el valor 34 a la variable x
set x 34

# Se asigna el valor contenido en x a la variable y
set y $x

# Se calcula la expresión matemática x+y, y se
# asigna el resultado a la variable z
set z [expr $x+$y]
```

Ejemplo 1.4 Definición de variables y asignación de valores.

[1] Programas escritos en lenguaje intérprete.
[2] Las instrucciones del código se van traduciendo una a una conforme se van ejecutando.

El comando *set* es usado para asignar un valor a una variable. El primer argumento es el nombre de la variable y el segundo argumento es el valor que debe tomar esa variable.

Anteponiendo el signo *$* al nombre de una variable, se puede obtener su valor.

OTcl trata a todas las variables como cadenas de caracteres (*strings*), y cuando es necesario las convierte en números (por ejemplo, para realizar operaciones matemáticas).

El comando *expr* permite indicar al intérprete que la cadena que sigue debe ser tratada como una expresión matemática.

La cadena de caracteres contenida entre corchetes, es considerada como un comando; dicha cadena se evalúa y se sustituye por el valor obtenido.

Comentarios

Para agregar un comentario dentro del *script* se antepone el símbolo #, como se observa en el Ejemplo 1.4.

Estructuras de Control

Los comandos de control de flujo son similares a sus equivalentes en C++.

En el Ejemplo 1.5 se presenta la sintaxis para el comando *if*; en el Ejemplo 1.6 se presenta la sintaxis para el comando *for* y en el Ejemplo 1.7 se presenta la sintaxis para el comando *while.*

- *Comando if*

```
# Dependiendo del valor que tenga la variable x, se realizará
# determinada acción, en este caso se imprimirá el respectivo mensaje
set x  1
if {$x ==0} {
# El comando puts permite imprimir en pantalla mensajes o
# valores de variables
puts "x es igual a 0"
} elseif {$x==1} {
puts "x es igual a 1"
} else {
puts "x es diferente de 0 y de 1"
}
# Como resultado de la ejecución del script, se imprimirá en
# pantalla el mensaje: x es igual a 1
```

Ejemplo 1.5 *Script* OTcl que presenta un ejemplo de uso del comando i*f*.

- *Comando for*

```
# El lazo se ejecutará mientras la variable i sea menor que 10,
# y se incrementará de 2 en 2 cada vez  que se cumpla la
# condición i<=10

for {set i 0} {$i<=10} {incr i 2} {
puts $ i
}

# Como resultado de la ejecución del script, se
# imprimirá en pantalla: 0 2 4 6 8 10
```

Ejemplo 1.6 *Script* OTcl que presenta un ejemplo de uso del comando *for*.

- *Comando while*

```
# Siempre que el valor de i sea menor ó igual a 3
# se imprimirá el valor de la variable y se
# incrementará en uno

set i  1
while {$i<=3} {
puts $i
incr i
}

# Como resultado de la ejecución del script, se imprimirá en
# pantalla: 1 2 3
```

Ejemplo 1.7 *Script* OTcl que presenta un ejemplo de uso del comando *while.*

- *Comando foreach*

El comando *foreach* asigna a una variable un elemento de una lista en cada paso (ver el Ejemplo 1.8).

```
# La variable x tomará los valores de la lista {1 2 3 5 9 10}
# en cada paso

foreach x {1 2 3 5 9 10} {
set y [expr $x*10]
puts $y
}

# Como resultado de la ejecución del script, se imprimirá en
# pantalla: 10 20 30 50 90 10
```

Ejemplo 1.8 *Script* OTcl que presenta un ejemplo de uso del comando *foreach*.

Procedimientos

Los procedimientos tienen el mismo objetivo que las funciones en C++.

Un procedimiento en OTcl es definido con el comando *proc* (ver el Ejemplo 1.9). El primer argumento es el nombre del procedimiento, el segundo argumento es el parámetro ó la lista de parámetros, y a continuación el cuerpo del procedimiento.

Al igual que en C++, toda variable creada dentro del procedimiento es local al mismo.

A diferencia de C++, no se requiere indicar el tipo de retorno del procedimiento.

Para acceder a una variable global desde un procedimiento, se utiliza el comando *global*, seguido del nombre de la variable, como se muestra en el Ejemplo 1.9.

```
# Se define un procedimiento Nuevo_proc en el que se utiliza una
# variable global a

set a 10

proc Nuevo_proc {x y} {
global a
set z [expr $x*$y]
return $a+$z
}

# A continuación se llama al procedimiento Nuevo_proc,
# enviándole como argumentos los valores 2 y 3, el resultado
# retornado se asigna a la variable a
set a [$Nuevo_proc 2 3]
# El valor asignado a la variable a será :16
```

Ejemplo 1.9 *Script* OTcl que presenta la declaración de procedimientos y el uso de variables globales.

Clases

Para declarar una nueva clase se utiliza la palabra reservada *Class,* seguida del nombre de la clase (ver el Ejemplo 1.10).

En el lenguaje OTcl, el constructor de la clase es el procedimiento *init{},* y el destructor es el procedimiento *destroy{}*.

Los procedimientos de una clase se los declara utilizando la palabra reservada *instproc,* la cual debe ser antecedida por el nombre de la clase y seguida por sus parámetros.

Para declarar variables de una clase se utiliza la palabra reservada *instvar, antecedida de $self* (puntero al mismo objeto, tiene el mismo concepto que el puntero *this*), y seguida del nombre de la variable. La declaración se la realizará en cada procedimiento de la clase que la utilicen, tomando como referencia el Ejemplo 1.10 se puede observar que se declara a la variable *varB1_* tanto en el constructor como en el procedimiento *procB*.

Para definir una clase derivada se utiliza la palabra reservada *–superclass* seguida del nombre de la clase de la cual va a heredar.

La creación de objetos se lo realiza con el comando *new*, seguido del nombre de la clase. Con el nombre del objeto creado se puede acceder tanto a los procedimientos como a las variables de la clase.

El Ejemplo 1.10 presenta un *script* OTcl en el que se definen dos clases y se instancian objetos de ellas.

```
#-------------------------------------
# Creación de la clase base
#-------------------------------------
Class Clase_Base

# Constructor de Clase_Base
Clase_Base instproc init {a} {

# Declaración de una variable de la clase Clase_Base
$self instvar varB1_

# Asignación de valor a la variable de la Clase_Base
set varB1_ $a
}


# Definición de un procedimiento de la clase Clase_Base
Clase_Base instproc procB {} {

# Utilización de la variable varB1_, declarada anteriormente en el
# constructor
$self instvar varB1_

# Declaración de una variable de la clase Clase_Base
$self instvar varB2_

# Se imprime los valores de varB1_ y varB2_
puts “Clase Base: los valore de (varB1_,varB2_)son
($varB1_,$varB2_)”

}
```

```
#-------------------------------------
# Creación de la clase derivada
#-------------------------------------
Class Clase_Derivada -superclass Clase_Base

# Constructor de Clase_Derivada
Clase_Derivada instproc init {a} {

# Declaración de la variable
$self instvar varD1_

# Asignación de valor
set varD1_ $a
}

# Definición de un procedimiento de la clase Clase_Derivada
Clase_Derivada instproc procD {} {

$self instvar varD1_
$self instvar varD2_

# Se imprime los valores de varD1_ y varD2_
puts “Clase Derivada: Los valores de (varD1_,varD2_)son
($varD1_,$varD2_)”
}


# Creación de un objeto de la Clase_Base
set objeto1 [new Clase_Base 5 ]
$objeto1 set varB2_ 10

# Creación de un objeto de la Clase_Derivada
set objeto2 [new Clase_Derivada 15]
$objeto2 set varD2_ 20

# Llamada al procedimiento proc1 para cada objeto
$objeto1 procB
$objeto2 procD

# Como resultado se obtendrá en pantalla:
# Clase Base: los valores de (varB1_,varB2_)son(5,10)
# Clase Derivada: los valores de (varD1_,varD2_)son(15,20)
```

Ejemplo 1.10 *Script* OTcl que presenta la definición y derivación de clases.

Comandos de Entrada / Salida

Para crear un nuevo archivo se utiliza el comando *open*, seguido del nombre del archivo y del modo de acceso; este comando devolverá un identificador para el archivo, con el cual se podrán realizar otras tareas, como: lectura, escritura y cierre.

Los modos de acceso para el archivo pueden ser:

- **r** : sólo lectura (el archivo debe existir).
- **r+** : lectura y escritura (el archivo debe existir).
- **w** : sólo escritura (si existe el archivo lo sobrescribe, caso contrario lo crea).
- **w+** : lectura y escritura (si existe el archivo lo sobrescribe, caso contrario lo crea).
- **a** : sólo escritura (el archivo debe existir, añade datos al final del archivo).
- **a+** : lectura y escritura (el archivo debe existir, añade datos al final del archivo).

Para cerrar el archivo se utiliza el comando *close* seguido de la variable que lo identifica.

En el Ejemplo 1.11 se presenta la utilización de los comandos *open* y *close*.

Comandos especiales

Se pueden ejecutar comandos externos al simulador con el comando *exec***,** seguido de la aplicación y del nombre del archivo con su respectiva extensión.

El comando *exit*, se utiliza para finalizar una aplicación, a éste comando le sigue un número, el cual retornará para indicar el estado del sistema; si el valor de retorno es cero, se indica al sistema que la salida ha sido exitosa.

En el Ejemplo 1.11 se presenta la utilización de los comandos antes descritos.

```
# Creación de un objeto ns de la clase Simulator
set ns [new Simulator]

# Creación del archivo out.nam, su identificador es asignado
# a la variable nf
set nf [open out.nam w]

# Invocación al procedimiento namtrace-all de la clase
# Simulator, que escribirá los eventos en el archivo out.man,
# para lo cual es necesario enviar como argumento la variable $nf
$ns namtrace-all $nf

# Se cierra el archivo nf
close $nf
# Ejecución la aplicación nam con el archivo out.nam
exec nam out.nam

# Finalización de la aplicación nam
exit 0
```

Ejemplo 1.11 *Script* OTcl en el que se hace uso de los comandos *open, close, exec y exit.*

1.3.2. JERARQUÍA

El simulador NS-2 presenta una jerarquía de clases en C++ (denominada jerarquía compilada) y una jerarquía de clases similar en OTcl (denominada jerarquía interpretada), la cual va a ser utilizada desde el intérprete OTcl. Las dos jerarquías están estrechamente relacionadas una con otra; desde la perspectiva del usuario, existe una correspondencia uno a uno entre las clases de las jerarquías **[2]**.

La jerarquía de clases interpretada es automáticamente creada a través de las funciones definidas en la clase *TclClass* que será explicada en la Sección 1.3.3.3.

Existen otras clases en el espacio C++ que no tienen correspondencia en el espacio OTcl, debido a que sirven para el funcionamiento interno del simulador, y por tanto no necesitan ser instanciadas en el intérprete OTcl. Del mismo modo, existen clases en el espacio OTcl que no tienen correspondencia en el espacio C++, ya que son utilizadas solo para la manipulación del simulador.

La Figura 1.7 presenta la jerarquía de clases compilada; a lo largo del documento se irán mencionando las clases relevantes.

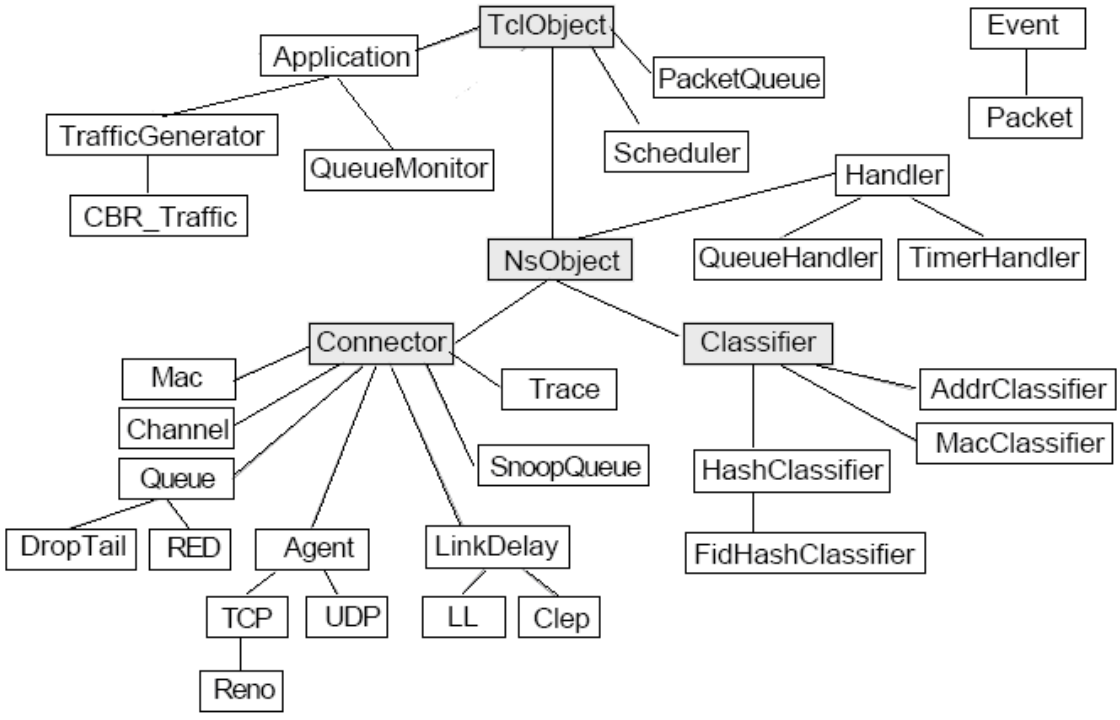


Figura 1.7 Jerarquía de clases compilada **[6]**.

1.3.3. CLASES PRINCIPALES DEL SIMULADOR NS-2

A continuación se describen las clases principales que son utilizadas para el funcionamiento del simulador, cabe indicar que éstas se encuentran implementadas en el espacio C++ pero no forman parte de la jerarquía de clases compilada, a excepción de la clase *TclObject* que forma parte de la jerarquía (ver la Figura 1.7). Cada una de estas clases está implementada independientemente (no pertenecen a una jerarquía común).

Su implementación se la puede encontrar en el directorio /ns-allinone2.30/tclcl-1.18 (ver el Anexo B).

1.3.3.1. Clase Tcl

Clase implementada en C++, que encapsula una instancia del intérprete OTcl y provee funciones para acceder y comunicarse con éste **[2]**.

A continuación se describen las principales operaciones que provee la clase *Tcl*:

1.3.3.1.1. Obtener una referencia a la instancia de la clase Tcl

A través de la función estática *instance()* se puede obtener una referencia a la variable miembro estática *tcl* de tipo *Tcl* y que servirá para acceder a otras funciones que ofrece la clase.

```
Tcl& tcl_prueba = Tcl::instance();
```

1.3.3.1.2. Invocar procedimientos de OTcl

A través de la instancia *tcl_prueba* se pueden invocar comandos OTcl desde el espacio C++, para ello se hace uso de las siguientes funciones:

- **eval(char *s):** Función que invoca a la función *Tcl_GlobalEval()* para ejecutar un comando OTcl desde el espacio C++ (en este caso el contenido de la variable *s*), como resultado el intérprete retornará un valor que puede ser *TCL_OK*[1]o *TCL_ERROR*[2] **[2]**.

[1] Indica que el comando se ejecutó de manera correcta.
[2] Indica que el comando no pudo ser ejecutado de manera satisfactoria.

- **evalc(const char *s):** Función que copia el comando OTcl (*s*) en un búfer interno y, a continuación invoca a la función antes descrita enviándole como argumento el contenido del búfer interno.
- **eval():** Función que ejecuta el comando almacenado en el búfer interno.

En el Ejemplo 1.12, se indica la utilización de las funciones antes descritas.

```
// Se obtiene una referencia a la instancia de la clase Tcl
Tcl& tcl_prueba = Tcl::instance();

char ejemplo[128];

strcpy(ejemplo,“ puts Mensaje de prueba de la función eval”);

// Se ejecuta el comando almacenado en la variable ejemplo
// mediante el uso de la función eval(char *s)
tcl_prueba.eval(ejemplo);

// Otra forma de utilizar la función eval(), es enviando
// directamente el comando OTcl en su argumento
tcl_prueba.eval(“ puts Mensaje de prueba de la función eval”);

// El comando “puts Mensaje de prueba de la función eval()
// sin enviar argumento” es almacenado en el buffer interno
// del intérprete
sprintf(tcl_prueba.buffer(),“puts Mensaje de prueba de la función
eval() sin enviar argumento”);

// Se ejecuta el comando almacenado en el buffer interno
tcl_prueba.eval()

// Se ejecuta el comando “puts Mensaje de prueba de la función
// evalc()”, y lo almacena en el buffer interno
tcl_prueba.evalc(“puts Mensaje de prueba de la función evalc()”);
```

Ejemplo 1.12 Utilización de las funciones para invocar procedimientos OTcl.

1.3.3.1.3. Retornar resultados hacia el intérprete y obtener resultados desde el intérprete

La variable miembro *result* de la clase *Tcl* almacena los resultados que se recogen desde el intérprete o se retornan hacia él, mediante las funciones *result(const char *s)* , *resultf(const char *s,…) y result(void),* respectivamente.

En el Ejemplo 1.13 y Ejemplo 1.14 se indica la utilización de las funciones antes mencionadas.

```
// Se pasan resultados hacia el intérprete
if (strcmp(argv[1], "now") == 0) {
tcl.resultf("%.17g", clock());
return TCL_OK;
}
tcl.result("Invalid operation specified");
return TCL_ERROR;
```

Ejemplo 1.13 Utilización de las funciones para retornar resultados hacia el intérprete **[2]**.

```
// Se ejecuta el comando Simulator set NumberInterfaces_
tcl.evalc("Simulator set NumberInterfaces_");

// El resultado de la ejecución del commando anterior se almacena en la
// variable ni, y con éste se determina si se ejecutó correctamente
char* ni = tcl.result();

// Si el valor obtenido como resultado es diferente de 1 se ejecuta el
// comando Simulator set NumberInterfaces_ 1
if (atoi(ni) != 1)
    tcl.evalc("Simulator set NumberInterfaces_ 1");
```

Ejemplo 1.14 Utilización de las funciones para obtener resultados desde el intérprete **[2]**.

1.3.3.2. Clase TclObject

Esta clase se encuentra implementada en el lenguaje C++ y en el lenguaje OTcl. Sirve como clase base, de la cual heredarán todas las clases que pueden ser manipuladas desde OTcl,

Es necesario indicar que la clase *TclObject* dentro del espacio OTcl, ha sido renombrada en las últimas versiones del simulador NS-2 como *SplitObject*.

Las principales operaciones que permite realizar esta clase son:

1.3.3.2.1. Exportar variables de las clases en C++ a OTcl

Se puede establecer un enlace bidireccional entre una variable miembro compilada y una variable miembro interpretada, de tal manera que el valor asignado a una variable sea el mismo en los dos espacios. El enlace debe ser establecido en el constructor de la nueva clase que se implemente en C++.

En NS-2 en el espacio OTcl se definen cinco tipos de datos (real, entero, tiempo, ancho de banda, booleano) que son especificados de diferente forma, por lo que se establecen cuatro diferentes funciones dentro de la clase *TclObject* para realizar el enlace bidireccional mencionado.

- void bind(const char* var, double* val)

- void bind(const char* var, int* val);
- void bind_bw(const char* var, double* val)
- void bind_time(const char* var, double* val)
- void bind_bool(const char* var, int* val)

En el Ejemplo 1.15 y Ejemplo 1.16 se presenta el enlace entre las variables en los dos espacios, y su asignación de valores **[2]**.

```
/* Establecimiento del enlace entre las variables en los
espacios C++ y OTcl*/
/*
 * Definición de las variables
double pdistance_;
int requestor_;
double lastSent_;
double ctrlLimit_;
int running_ ;
*/
/* Constructor de la clase Protocolo */
Protocolo::Protocolo()
{
// Variable real
bind("pdistance_", &pdistance_);

// Variable entera
bind("requestor_", &requestor_);

// Variable de tiempo
bind_time("lastSent_", &lastSessSent_);


// Variable de ancho de banda
bind_bw("ctrlLimit_", &ctrlBWLimit_);

// Variable booleana
bind_bool("running_", &running_);
}
```

Ejemplo 1.15 Enlace entre las variables de una clase en C++ y su correspondiente en OTcl.

```
# Asignación  de valores en el espacio OTcl
# Protocolo es la clase reflejada en el espacio OTcl creada
# a partir de la clase Protocolo en C++
Protocolo set pdistance_ 15.0
Protocolo set requestor_ 10.0
Protocolo set lastSessSent_ 8.345m
Protocolo set ctrlBWLimit_ 1.44M
Protocolo set running_ f
```

Ejemplo 1.16 Asignación de valores a las variables en el espacio OTcl.

1.3.3.2.2. Exportar funciones de las clases en C++ a OTcl

Desde el espacio OTcl se pueden crear objetos de las clases de la jerarquía OTcl, los mismos que son reflejados en el espacio C++; sin embargo, no se puede invocar directamente a las funciones implementadas en C++ para ese objeto por lo que se establece un mecanismo indirecto de invocación

En la Figura 1.8 se presenta el procedimiento que se sigue para exportar funciones de las clases en C++ a OTcl, para lo que se ha utilizado el objeto *p0* de la clase Agent/Protocolo.

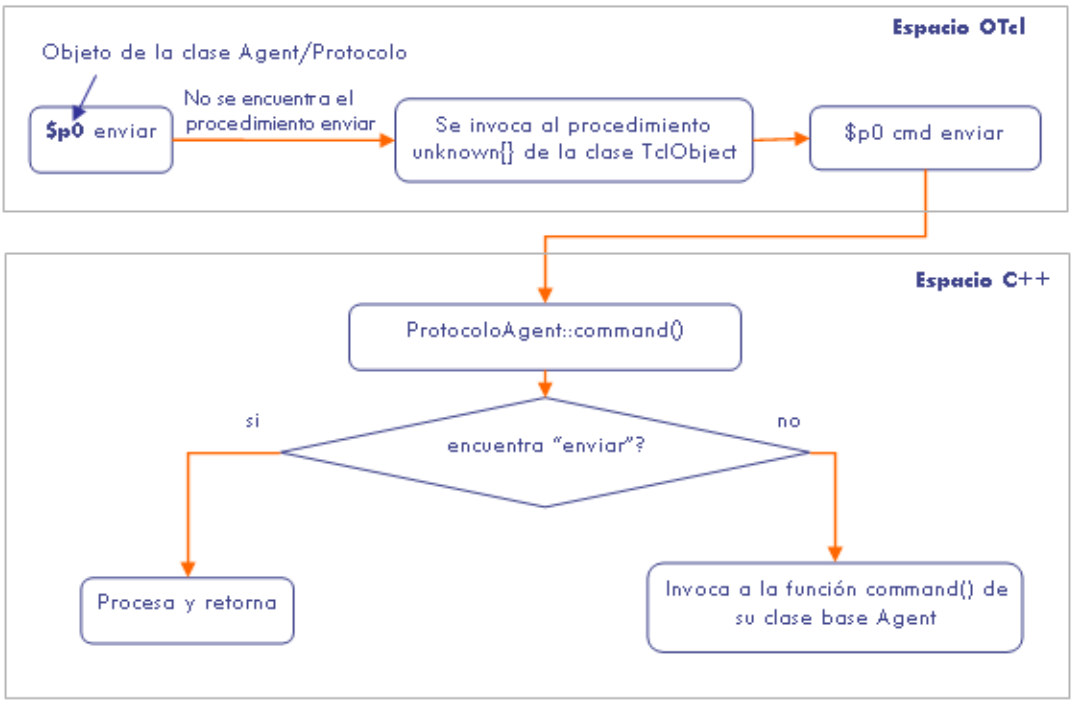


Figura 1.8 Procedimiento para exportar funciones de las clases en C++ a OTcl.

Es necesario mencionar que en proceso de vinculación entre los espacios C++/OTcl (explicado en la Sección 3.3.1) se establece el procedimiento *cmd{}* para cada una de las clases de jerarquía interpretada, cuya finalidad se la explica a continuación.

En la Figura 1.8, se puede apreciar que con el objeto *p0* se intenta invocar al procedimiento *enviar,* el cual no es encontrado, por tanto se invoca al procedimiento *unknown{}* de la clase base *TclObject.* El procedimiento *unknown{}* invocará al procedimiento *cmd{}* de la clase Agent/Protocolo, enviándole como argumento el nombre del procedimiento *enviar*. El procedimiento *cmd{}* invocará a la función *command()* (ver el Ejemplo 1.17) del objeto reflejado en C++, el cual recoge de sus argumentos el nombre del procedimiento buscado en OTcl (*enviar*)

e ingresa a verificar si para este nombre se debe realizar alguna operación, caso contrario invocará a la función *command()* de la clase base.

```
/* Implementación de la función command()*/

// El argumento argc indica el número de argumentos
// especificados en la línea de comandos para el intérprete
// El argumento argv contiene la siguiente información
// argv[0] contiene el nombre del procedimiento cmd
// argv[1] contiene el nombre del procedimiento (enviar)invocado en el
// OTcl espacio

int ProtocoloAgent::command(int argc, const char*const* argv)
{
if (argc == 2) {

// Si el  nombre del procedimiento buscado en OTcl coincide con el
// string send entonces se invoca a la función enviar()
    if (strcmp(argv[1], "enviar") == 0) {
          enviar();   // Función de la clase ProtocoloAgent
    return (TCL_OK);

    if (strcmp(argv[1], "detener") == 0) {
          detener();  // Función de la clase ProtocoloAgent
    return (TCL_OK);

    }
  }
return (Agent::command(argc, argv));
}
```

Ejemplo 1.17 Implementación de la función *command()* de la clase *ProtocoloAgent*.

1.3.3.3. Clase TclClass

Es una clase abstracta implementada en lenguaje C++; las clases derivadas a partir de ésta ofrecen dos funcionalidades:

- Proveer mecanismos para construir la jerarquía interpretada, la cual reflejará a la jerarquía compilada.
- Proveer mecanismos para instanciar nuevos objetos de red (dentro del espacio OTcl), y asociarlos a su correspondiente objeto en el espacio C++.

Las funciones de esta clase son utilizadas para el proceso de vinculación, por tanto serán explicadas en la Sección 3.3.

1.3.4. VISUALIZACIÓN DE LOS RESULTADOS

Los resultados del simulador NS-2 son comúnmente presentados en dos tipos de archivos .tr y .nam, que contienen la misma información pero en distinto formato. En la Figura 1.9 se presentan los archivos de salida.

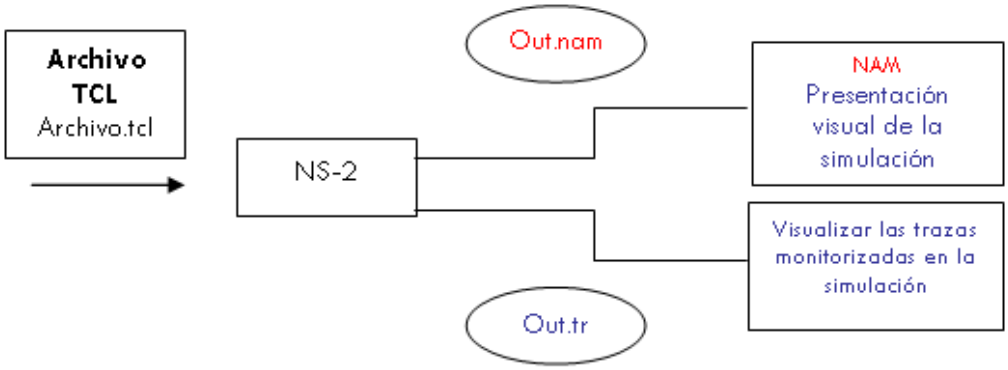


Figura 1.9 Archivos de salida del simulador NS-2 **[3]**.

1.3.4.1. Archivos .nam

NS-2 hace uso de la aplicación NAM (*Network AniMator*) para presentar gráficamente la topología física de la red, como también para visualizar dinámicamente los resultados obtenidos de la simulación.

Existen dos maneras para ejecutar esta aplicación:

1. Desde la línea de comandos:

```
nam out.nam
```

2. Desde el *script* de simulación (archivo .tcl):

```
exec nam out.nam
```

En donde *out.nam* es un archivo de trazas en formato *nam* generado por NS-2 como resultado de la simulación.

En la Figura 1.10 se presenta la interfaz gráfica de la aplicación NAM.

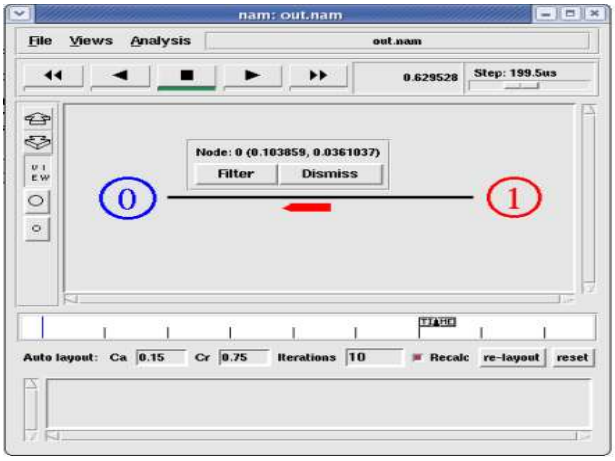


Figura 1.10 Interfaz gráfica de la aplicación NAM.

Controles de la interfaz gráfica NAM

En la Figura 1.11 se indican los controles con las que cuenta la interfaz gráfica NAM.

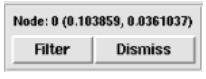
	Ejecutar animación
	Detener animación
	Ejecutar la animación hacia atrás
	Adelantar la animación en un tiempo de 25 pasos
	Retroceder la animación en un tiempo de 25 pasos
	Visor que indica el tiempo actual de la animación
	Cursor para fijar el tiempo de los paso de animación
	Control que permite modificar el tamaño de los nodos
	Control que permite aumentar el tamaño de los nodos
	Control que permite disminuir el tamaño de los nodos
	Aumentar zoom
	Disminuir zoom
	Barra que marca la evolución del tiempo
	Visor que indica el nodo y su posición

Figura 1.11 Controles de la interfaz gráfica de la aplicación NAM

1.3.4.2. Archivos .tr

Al observar el contenido de este archivo, se distinguen todos los eventos registrados durante la simulación, uno en cada línea.Formato de los eventos en el archivo .tr:

Evento	Tiempo	Nodo Fuente	Nodo Destino	Tipo de Paquete	Tamaño Paquete	Bandera	Id. Flujo	Dir. Origen	Dir. Destino	Número Secuencia	Id. Paquete
-, +, r, d	double	Int	int	string	int	string	int	Int,int	Int,int	int	int
-	0.32	0	1	frame	1024	-------	0	0.0	1.0	1	1

Los eventos que se pueden presentar son:

- **-** : En la cola de espera
- **+** : Sale de la cola de espera
- **r** : Recibido
- **d** : Descartado

Cabe mencionar que tanto en la dirección origen como en la destino, se indican el nodo y el puerto (por ejemplo 0.0).

1.3.5. EJEMPLO DE UN SCRIPT DE SIMULACIÓN

En el Ejemplo 1.18 se presenta un *script* de simulación OTcl, en el cual se hace uso de la sintaxis del lenguaje OTcl explicada anteriormente. Además, se hace uso de algunos procedimientos de la clase *Simulator*, que serán explicados en el Capítulo 3.

```
# Creación de un objeto ns de la clase Simulator
set ns [new Simulator]

# Creación del archivo out.nam, su identificador es asignado
# a la variable nf
set nf [open out.nam w]
# Invocación al procedimiento namtrace-all de la clase
# Simulator, pasándole como argumento la variable $nf
$ns namtrace-all $nf

# Creación del archivo out.tr, su identificador es asignado
# a la variable ntr
set ntr [open out.tr w]
# Invocación al procedimiento trace-all de la clase
# Simulator, pasándole como argumento la variable $ntr
$ns trace-all $ntr

# Definición del procedimiento finish
proc finish {} {
```

```
    # Acceso a las variables globales ns, nf, ntr
          global ns nf ntr
    # Invocación al procedimiento flush-trace de la
    # clase Simulator
          $ns flush-trace
    # Se cierra el archive nf
          close $nf
    # Se cierra el archive ntr
          close $ntr
    # Ejecución del archivo out.nam, con la aplicación nam
          exec nam out.nam
    # Finalización de la aplicación nam
          exit 0
    }
    #=======================================#
    # Definición de la topología física     #
    #=======================================#
    # Creación de los objetos n0, n1 de la clase node
    set n0 [$ns node]
    set n1 [$ns node]
    # Creación del enlaces entre los nodos n0,n1
    $ns duplex-link $n0 $n1 1Mb 20ms DropTail
    #=======================================#

    # Invocación al procedimiento finish al tiempo 2 seg.
    $ns at 2 "finish"
    # Ejecución de la simulación
    $ns run
```

Ejemplo 1. 18 *Script* de simulación escrito en lenguaje OTcl.

A continuación, se presenta la Figura 1.12 en la que se pueden observar los resultados obtenidos a partir de la ejecución del *script* indicado en el Ejemplo 1.18.

En la interfaz gráfica de la aplicación NAM se presenta la simulación de dos nodos que se conectan a través de un enlace *duplex*; sin embargo, en éste ejemplo los nodos no realizan ninguna acción.

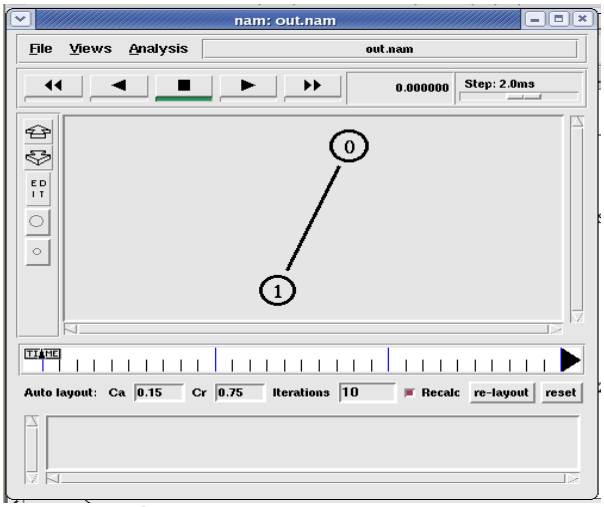


Figura 1. 12 Ejecución del archivo out.nam con la aplicación NAM.

En cuanto al archivo *out.tr*, en éste no se registra ningún evento, ya que no se configuró un protocolo que se ejecute sobre los nodos.

CAPÍTULO 2

IMPLEMENTACIÓN EN OMNET++

Todos los elementos (protocolos, dispositivos, conexiones, etc.) que intervienen en un escenario de simulación de red son considerados como módulos, los cuales pueden ser simples o compuestos (aspectos mencionados en la Sección 1.2.1).

IMPLEMENTACIÓN DE UN PROTOCOLO EN OMNET++

Para llevar a cabo la implementación de un nuevo protocolo en OMNET++ es necesario tener una visión clara de la funcionalidad de las clases de la que dependerá la nueva implementación, es decir, de las clases *cModule* y *cSimpleModule* como se observa en la Figura 2.1.

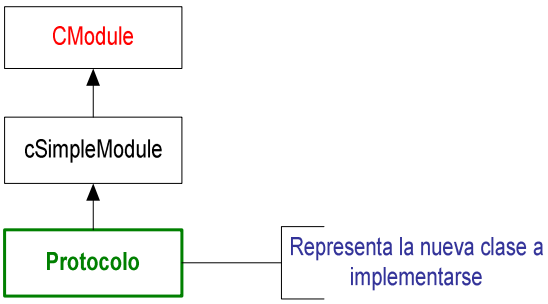


Figura 2.1 Ubicación en la jerarquía de clases de una clase que representa a un nuevo protocolo.

Clase cModule

Es la clase base para la clase *cSimpleModule.* Las principales funciones que ésta provee son:

- **initialize():** Función virtual que se encarga de obtener los diferentes parámetros configurados en el archivo escrito en lenguaje NED o indicados en el archivo de inicialización omnetpp.ini, asignarlos a los datos miembro correspondientes en lenguaje C++. Esta función es invocada por el simulador para cada uno de los módulos, una vez que han sido creados.
- **finish():** Es una función virtual invocada cuando la simulación finaliza satisfactoriamente; es usada principalmente para recopilar resultados escalares y almacenarlos en un archivo con extensión .sca.

Clase cSimpleModule

La finalidad de la clase es proveer funciones que permitan programar y manipular las unidades de datos[1], es así por ejemplo se las puede crear y/ó destruir.

A través de la implementación de una clase que se derive de *cSimpleModule* se puede especificar el comportamiento de un protocolo. La principal función virtual que debe ser redefinida en las clases derivadas es:

- **handleMessage():** Función en la que se debe implementar el comportamiento de un protocolo ante la llegada de información. El *kernel* del simulador invocará esta función cuando el módulo reciba una unidad de datos.

Adicionalmente, *cSimpleModule* provee funciones tales como: *send(), scheduleAt(), simTime(), sendDelayed()* y otras que serán explicadas en el transcurso de la implementación de los protocolos.

En la Figura 2.2 se presenta como el módulo simple hace uso de las funciones *handleMessage()* y *send()* para la recepción y envío de las unidades de datos.

[1] Objetos de la clase *cMessage* o de alguna derivada de ésta.

Figura 2.2 Comportamiento de un módulo simple.

1.4. IMPLEMENTACIÓN DE PROTOCOLOS ELEMENTALES DE ENLACE DE DATOS

Los protocolos que se implementan a continuación se presentan de forma secuencial de acuerdo a un grado de complejidad creciente, lo que requiere la introducción de nuevas funciones y variables; por tanto, se ha considerado que en cada protocolo se explicará únicamente lo que se agregue a la implementación respecto al protocolo anterior.

1.4.1. PROTOCOLO SIMPLEX SIN RESTRICCIONES

1.4.1.1. Descripción del protocolo

En este protocolo se implementan dos de las posibles funciones que ofrece la capa de enlace de datos, que son: proporcionar servicios a la capa de red y entramado (ver el Anexo C).

Además, el protocolo "*simplex* sin restricciones" tiene las siguientes características **[7]**:

- Los datos se transmiten sólo en una dirección.
- La capa de red del emisor siempre está lista para enviar información a la capa de enlace de datos, de igual forma la capa de red del receptor siempre está lista para recibir información de la capa de enlace de datos.
- El tiempo de procesamiento puede ignorarse.
- Existe un espacio infinito de búfer.
- El canal de comunicación nunca tiene problemas ni pierde tramas.

1.4.1.2. Diseño e implementación en C++

Para implementar este protocolo es necesario definir una unidad de datos y dos módulos simples (emisor y receptor) que la procesen.

A continuación, la Figura 2.3 presenta el diagrama de flujo para el módulo emisor en donde se puede observar que éste está listo para enviar datos; en la Figura 2.4 se presenta el diagrama de flujo para el módulo receptor en donde se puede observar que siempre está listo para recibir datos.

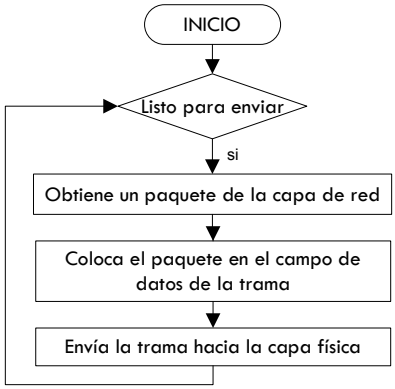


Figura 2.3 Diagrama de flujo para el módulo emisor.

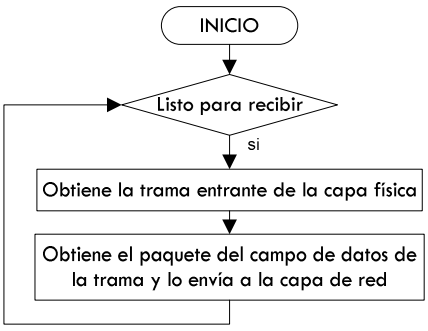


Figura 2.4 Diagrama de flujo para el módulo receptor.

1.4.1.2.1. Unidad de datos

En OMNET++, la unidad fundamental para intercambio de información es un objeto de la clase *cMessage* o de alguna clase derivada de ésta.

Para representar la unidad de datos para un determinado protocolo se debe implementar una clase derivada de la clase *cMessage*.

Clase cMessage

La clase *cMessage* forma parte de la librería del *kernel* de simulación; a continuación se describen sus principales variables:

- **msgkind:** Variable entera que puede ser configurada por el usuario para identificar determinado tipo de mensaje.
- **len:** Variable entera que representa el tamaño en bits de una unidad de datos.
- **error**: Variable utilizada como bandera de error.

La clase *cMessage* provee un constructor por defecto y varios con argumentos. Generalmente se utiliza el constructor con argumentos en el que se indica el nombre del mensaje y el tipo. En el Código 2.1 se presenta la utilización de los constructores.

```
// Constructor por defecto
cMessage *msg1 = new cMessage();

// Constructor que recoge en el argumento el nombre del mensaje
cMessage *msg2= new cMessage("frame");

// Constructor que recoge en el argumento el nombre del mensaje
// y el tipo de mensaje
cMessage *msg3= new cMessage("frame",1);
```

Código 2.1 Constructores para la clase *cMessage*.

Las principales funciones que permite manipular las variables antes descritas son:

- **setKind():** Función que permite configurar la variable *msgkind*.
- **setLength():** Función que permite configurar la variable *len*.
- **setBitError():** Función que permite configurar la variable *error*.
- **kind():** Función que retorna el valor de la variable *msgkind*.
- **hasBitError():** Función que retorna el valor de la variable *error***.**

Clase frame

Para los nuevos protocolos a desarrollarse se ha definido el formato de una unidad de datos representada mediante la clase *frame,* la misma que contiene los campos mostrados en la Figura 2.5. La clase *frame* es una clase derivada de la clase *cMessage.*

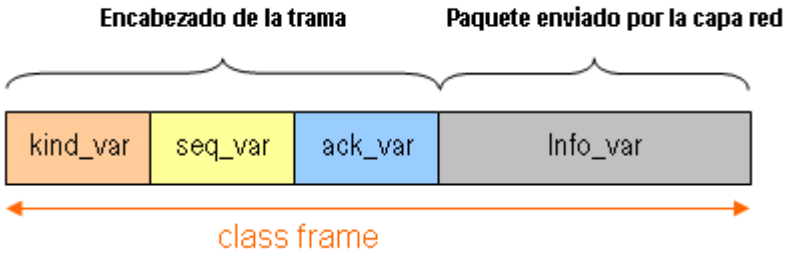


Figura 2.5 Campos de la trama.

OMNET++ ofrece al usuario la posibilidad de poder crear una clase derivada de la clase *cMessage* de una forma mucho más sencilla, a través de archivos .msg los cuales son convertidos automáticamente a código C++ mediante el compilador opp_msgc.

Para representar la trama de datos que será utilizada en éste y posteriores protocolos se ha creado el archivo *frame.msg,* el cual contiene la estructura de la trama. En el Código 2.2 se presenta el contenido del archivo *frame.msg.*

```
// Incluye el archivo packet.h
cplusplus {{
#include "packet.h"
}};

// Indica al compilador de mensajes que packet es de tipo struct
struct packet;

// Definición de un enumerado con el que se identificará
// el tipo de trama
enum frame_kind
{
     data = 1;// Si la trama es de datos
     ack = 2; // Si la trama es de control

};

// Definición del mensaje frame
// Los nombres de los campos de la trama se ajustan
// a la nomenclatura del libro Redes de Computadoras del autor
// A. Tanenbaum
```

```
message frame
{
     fields:
           // kind tomará valores del enumerado frame_kind
           int   kind enum(frame_kind); // Tipo de trama
           int   seq;    // Número de secuencia
           int   ack;    // Número de confirmación
           packet info;  // Campo de datos

};
```

Código 2.2 Definición del mensaje *frame* que representa el formato de la trama.

Luego de la compilación del archivo frame.msg se crean los archivos frame_m.h y frame_m.cpp que contienen la definición e implementación de la clase *frame.* En el Código 2.3 se presenta parte del contenido del archivo frame_m.h.

```
class frame : public cMessage
{
  protected:
    int kind_var;
    int seq_var;
    int ack_var;
    packet info_var;
    bool operator==(const frame&);

  public:
    frame(const char *name=NULL, int kind=0);
    frame(const frame& other);
    virtual ~frame();
    frame& operator=(const frame& other);
    virtual cPolymorphic *dup() const {return new frame(*this);}
    virtual void netPack(cCommBuffer *b);
    virtual void netUnpack(cCommBuffer *b);
    virtual int getKind() const;
    virtual void setKind(int kind_var);
    virtual int getSeq() const;
    virtual void setSeq(int seq_var);
    virtual int getAck() const;
    virtual void setAck(int ack_var);
    virtual packet& getInfo();
    virtual void setInfo(const packet& info_var);
};
```

Código 2.3 Definición de la clase *frame.*

Variables

Las variables que se generan para esta clase tienen correspondencia a las variables definidas en la sección *fields* de *message frame*, por ejemplo para la variable *seq* su correspondiente es *seq_var*.

Funciones

Se generan dos tipos de constructores: uno por copia y otro con argumentos, en el que se indica el nombre del mensaje y el tipo de mensaje.

Para las variables miembro de la clase, se generan funciones miembro públicas que permiten establecer u obtener sus valores. Además, se redefinen otras cuya funcionalidad será explicada de acuerdo al requerimiento de los protocolos a implementarse.

1.4.1.2.2. Módulo

Como se explicó anteriormente, los módulos simples en OMNET++, al ser entidades de procesamiento, pueden ser utilizados para representar protocolos y es aquí en donde se implementa funcionalidad para la creación, procesamiento y destrucción de la unidad de datos.

De acuerdo a la funcionalidad del protocolo "*simplex* sin restricciones", se ha considerado implementar dos módulos simples; un módulo emisor, representado por la clase *Protocolo1Sender* y un módulo receptor, representado por la clase *Protocolo1Receiver*.

Tanto la clase *Protocolo1Sender* como *Protocolo1Receiver,* son clases derivadas de la clase *cSimpleModule.*

Clase Protocolo1Sender

El objetivo de esta clase es enviar tramas al receptor sin ninguna limitación, por tanto la clase provee funcionalidad para: obtener un paquete de la capa de red, construir una trama y enviar la trama a su destino.

En el Código Protocolo 1.1 se presenta la definición de la clase *Protocolo1Sender*, en donde se presentan las variables y funciones que son propias para la representación del protocolo, así como también aquellas que son necesarias para la simulación en OMNET++.

Cabe mencionar que tanto los nombres de variables como de funciones se encuentran de acuerdo a la nomenclatura de la referencia **[7].**

```
#include <stdio.h>
#include <omnetpp.h>
#include "frame_m.h"
/*=========================================================*/
/*                     Protocolo1Sender                    */
/*=========================================================*/
class Protocolo1Sender:public cSimpleModule
{
private:
// Variables del protocolo
     frame *s;               // Variable que representa a la trama
                             // que será enviada
     packet buffer;          // Paquete que se obtiene  de la
                             // capa de red
     int frameSize;          // Tamaño de la trama
     int npacket;            // Representa el número de paquetes
                             // que se obtendrá de la capa de red
public:
// Constructor
     Protocolo1Sender();
// Destructor
     virtual ~Protocolo1Sender();
// Funciones del protocolo
     void from_network_layer(packet*  msg);    // Obtiene un paquete
                                               // de la capa de red
     void to_physical_layer(frame *psend);     // Envía una trama
                                               // hacia la capa física
     void crear_paquete(packet *msg);          // Genera un paquete en la
                                               // capa de red
// Función auxiliar
void registro_evento();      // Imprime en pantalla el nombre del módulo
                             // y el tiempo en que ocurre un evento
protected:
// Función de la clase cSimpleModule que se redefine
     virtual void initialize();    // Función que es invocada por
                                   // el kernel al iniciar la simulación
};
```

Código Protocolo 1.1 Definición de la clase *Protocolo1Sender*.

Variables

- **s:** Variable que representa a la trama que será enviada hacia el receptor.
- **buffer:** Variable de tipo *packet* que representa la información obtenida de la capa red, la cual será insertada dentro del campo de datos de la trama.
- **frameSize:** Representa el tamaño de la trama; este valor deberá ser configurado por el usuario en el archivo omnetpp.ini**.**
- **npacket:** Variable con la que se configura el límite del flujo de tramas. Su uso es sólo para efectos de la simulación pues la naturaleza del protocolo es enviar un flujo infinito de tramas.

Es necesario indicar que se hará uso de objeto global *ev* de clase *cEnvir* para utilizar el operador << que permite imprimir mensajes en pantalla.

Funciones

- **Protocolo1Sender():** Constructor que será invocado por el *kernel* del simulador en el proceso de creación del modelo, y por tanto no tiene argumentos, ya que nunca será instanciado directamente por el usuario.
- **~Protocolo1Sender():** Destructor virtual; aquí se deberá liberar la memoria reservada para los objetos que han sido creados dinámicamente por el operador *new* a través del operador *delete,* no se aplica en este ejemplo.
- **from_network_layer():** Función que en conjunto con *to_network_layer()* (función de la clase *Protocolo1ReceiverAgent*) representan la interfaz entre la capa de red y la capa de enlace de datos. En esta función, la capa de enlace de datos obtiene un paquete de la capa de red mediante la llamada a la función *crear_paquete().* Su implementación se presenta en el Código Protocolo 1.2.

```
void Protocolo1Sender::from_network_layer(packet *msg)
{
     ev<<" La capa de red envía un paquete "<<endl;
     crear_paquete(msg);
}

void Protocolo1Sender::crear_paquete(packet *msg)
{
      msg->data=" Nuevo Paquete";
      return;
}
```

Código Protocolo 1.2 Implementación de la función *from_network_layer()* y *crear_paquete()* de la clase *Protocolo1Sender.*

- **to_physical_layer():** Función en donde la capa de enlace de datos envía una trama a la capa física. Para su implementación se ha utilizado la función *send*() definida en la clase *cSimpleModule* cuya finalidad es enviar la trama *psend* por la compuerta[1] de salida *salida***.** Su implementación se presenta en el Código Protocolo 1.3.

[1] Representa el punto de conexión entre módulos.

```
void Protocolo1Sender::to_physical_layer(frame *psend)
{
     ev<<" Envía una nueva trama a la capa física"<<endl;
     // Función definida en la clase cSimpleModule
     send(psend,"salida");
}
```

Código Protocolo 1.3 Implementación de la función *to_physical_layer()* de la clase *Protocolo1Sender.*

- **registro_evento():** Función que es invocada para imprimir el nombre del módulo y el tiempo en que ocurre un evento sobre él. En esta función se hace uso de las funciones: *name()* (función definida en la clase *cObject*[1]) y *simTime()* (función definida en la clase *cSimpleModule*).

  La función *name()* devuelve un puntero al nombre del objeto, es decir, devuelve un puntero al nombre del módulo que ha sido creado en el lenguaje NED.

  La función *simTime()* permite obtener el tiempo actual (en segundos) de la simulación.

  En el Código Protocolo 1.4 se presenta la implementación de la función *registro_evento()* en donde de multiplica por 1000 el valor obtenido de la función *simTime()* para transformarlo en milisegundos.

```
void Protocolo1Sender::registro_evento()
{
     ev<<"-------------------------------------------------"<<endl;
     ev<<"NODO:"<<name()<<","<<"TIME:"<<simTime()*1000<<" ms"<<endl;

}
```

Código Protocolo 1.4 Implementación de la función *registro_evento()* de la clase *Protocolo1Sender*.

- **initialize()**: Función redefinida de la clase *cSimpleModule*, la cual es invocada en la creación de la topología (todos los módulos y conexiones del modelo). En ella se inicializan las variables y parámetros del módulo y, si la simulación lo requiere, se crea un evento inicial. En este caso, se ha implementado funcionalidad que simula la transmisión de datos infinita, para lo que se hace uso de la sentencia *for,* en donde, por efectos de visualización de resultados se ha fijado un límite. En el cuerpo de la sentencia *for* se implementa la funcionalidad de entramado, y además se

[1] Clase base de la clase *cModule*.

asigna el valor *frameSize* (valor en *bits*) a la variable *len* de la clase *cMessage* y finalmente se procede a enviar la trama a la capa física.

La función *par()* (función definida en la clase *cModule*) recoge como argumento la cadena de caracteres con la que se ha definido el parámetro en lenguaje NED (cuyo valor se ha configurado en el archivo omnetpp.ini). A través de esta función se puede hacer la correspondencia entre las variables definidas en C++ y NED. Es así que los parámetros *npacket* y *frameSize* (definidas en NED) tienen su correspondencia con las variables *npacket* y *frameSize* (definidas en C++), respectivamente.

En el Código Protocolo 1.5 se presenta la implementación de la función *initialize*().

```
void Protocolo1Sender::initialize()
{

// Se toma los valores configurados en el archivo de
// inicialización .ini o en el archivo .ned

     npacket= par ("npacket");
     frameSize= par ("frameSize");

     for(int i=0;i<npacket;i++)
     {
           // Función que imprimirá en pantalla
           // el nombre del nodo y el tiempo
           registro_evento();

           // Se asigna un espacio de memoria para el puntero s
           s=new frame();

           // Se obtiene un paquete de la capa de red
           from_network_layer(&buffer);

           // Se coloca el paquete en el campo info de la trama
           s->setInfo(buffer);

           // Se asigna el tamaño de la trama de acuerdo al valor
           // configurado en el archivo .ini o .ned
           s->setLength(frameSize);

           // Se envía la trama a la  capa física
           to_physical_layer(s);
     }
}
```

Código Protocolo 1.5 Implementación de la función *initialize()* de la clase *Protocolo1Sender*.

Clase Protocolo1Receiver

La funcionalidad de esta clase consiste en estar siempre listo para recibir las tramas enviadas por el emisor, para posteriormente obtener la información y enviarla a la capa de red.

En el Código Protocolo 1.6 se presenta la definición de la clase *Protocolo1Receiver*.

```
/*=========================================================*/
/*                     Protocolo1Receiver                  */
/*=========================================================*/
class Protocolo1Receiver:public cSimpleModule
{
private:

public:
// Constructor
     Protocolo1Receiver();
// Destructor
     virtual ~Protocolo1Receiver();
// Función del protocolo
     void to_network_layer(packet *msg);// Envía un paquete a
                                        // la capa de red
// Función auxiliar
     void registro_evento(); // Imprime en pantalla el nombre del nodo
                             // y el tiempo en que ocurre un
                             // evento
protected:
// Función de la clase cSimpleModule que se redefine
     virtual void handleMessage(cMessage *p_r_); // Función que recibe
                                                 // la trama desde la
                                                 // capa física
};
```

Código Protocolo 1.6 Definición de la clase *Protocolo1Receiver*.

Funciones

- **to_network_layer():** Función que permite enviar a la capa de red el paquete recibido en el campo de datos de la trama.

  En Código Protocolo 1.7 se presenta la implementación de la función *to_network_layer().*

```
void Protocolo1Receiver::to_network_layer(packet *msg)
{
     ev<<" La capa de red recibe:"<<msg->data<<endl;
}
```

Código Protocolo 1.7 Implementación de la función *to_network_layer()* de la clase *Protocolo1Receiver*.

- **handleMessage():** Función redefinida de la clase *cSimpleModule,* la cual será invocada automáticamente ante la llegada de una unidad de datos para posteriormente procesarla.

  Una vez recibida la unidad de datos mediante el puntero *p_r_* de tipo *cMessage*, es necesario convertirlo a un puntero de tipo *frame* para poder acceder a sus campos, para lo que se ha utilizado el *template check_and_cast<>()* provisto por OMNET++.

  El *template check_and_cast<>()* hace uso del operador *dynamic_cast<>* ofrecido por C++ y, lanza una excepción si es un puntero a NULL.

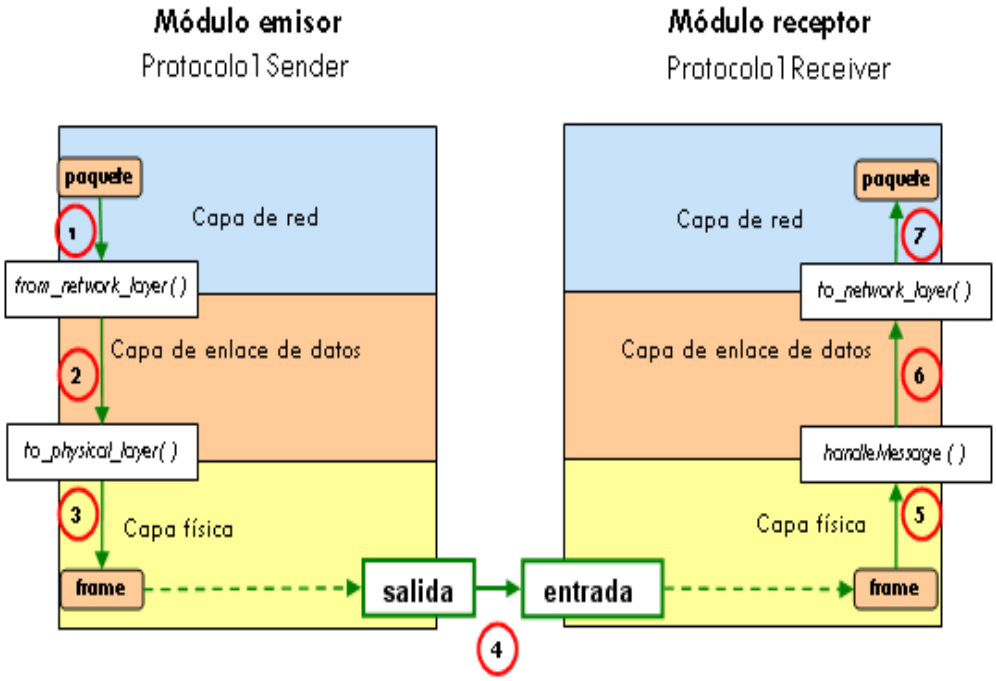
  Una vez obtenida la trama se procede a desencapsular el paquete contenido en el campo *info_var* y se elimina la trama.

  El Código Protocolo 1.8 presenta la implementación de la función *handleMessage().*

```
void Protocolo1Receiver::handleMessage(cMessage *p_r_)
{
     // Se imprime en pantalla
     // el nombre del módulo y el tiempo
     registro_evento();

     // Convierte el objeto recibido de tipo cMessage a tipo frame
     frame *r = check_and_cast<frame*>(p_r_);

     // Se envía los datos hacia la capa de red
     to_network_layer(&r->getInfo());

     // Se borra la trama recibida
     delete r;

}
```

Código Protocolo 1.8 Implementación de la función *handleMessage()* de la clase *Protocolo1Receiver*.

En la Figura 2.6 se presenta gráficamente la funcionalidad del protocolo "*símplex* sin restricciones".



1.- La función *from_network_layer()* recibe un paquete de la capa de red.
2.- El paquete recibido se encapsula en una trama.
3.- La función *to_physical_layer()* envía la trama creada hacia la capa física.
4.- Se envía la trama desde en módulo emisor hacia el módulo receptor
5.- La función *HandleMessage ()* del módulo receptor recibe la trama.
6.- Se desencapsula la trama y se obtiene el paquete.
7.- La función *to_network_layer()* envía el paquete a la capa de red.

El emisor enviará un infinito número de paquetes, para los cuales se repite el proceso antes detallado

Figura 2.6 Proceso que sigue el protocolo "*simplex* sin restricciones".

1.4.1.2.3. Define_Module(Protocolo1Sender)

Define_Module() es una macro[1] definido por la librería de OMNET++, es decir, que el compilador sustituirá esa línea por una serie de líneas de código previamente definidas. Esta instrucción la utiliza internamente OMNET++ para construir el modelo. El parámetro debe coincidir siempre con el nombre que se le ha dado al módulo en la declaración en NED (Define_Module(Protocolo1Sender)), y debe ser invocado en el archivo con extensión .cpp.

[1] Un macro es una plantilla de código parametrizada.

1.4.1.3. Configuración del escenario de simulación

1.4.1.3.1. Configuración del escenario de simulación en el lenguaje NED

```
// Definición del módulo emisor
simple Protocolo1Sender
    parameters:
        frameSize: numeric,        // Tamaño de la trama

        tamaniobuffer: numeric;    // Representa el número de paquetes
                                   // que se obtendrá de la capa de red
    gates:
        out: salida;               // Compuerta de salida
endsimple

// Definición del módulo receptor
simple Protocolo1Receiver
    gates:
        in: entrada;               // Compuerta de entrada
endsimple

// Definición de un módulo compuesto formado de
// los módulos simples emisor y receptor
module Protocolo1
    parameters:
        frameSize: numeric,
        npacket: numeric,
        propagacion: numeric,      // Tiempo de propagación
        vtx: numeric;              // Velocidad de transmisión

    submodules:

        A: Protocolo1Sender; // Instancia del módulo Protocolo1Sender
            parameters:
                npacket = npacket,
                frameSize = frameSize;
            // Se muestra la posición y el dispositivo que representa
            // al módulo A en el escenario de simulación
            display: "p=35,96;i=device/pc,cyan";

        B: Protocolo1Receiver;// Instancia del módulo Protocolo1Receiver
            // Se muestra la posición y el dispositivo que representa
            // al módulo B en el escenario de simulación
            display: "p=224,96;i=device/pc,gold";

    connections:
        // Configuración de los parámetros del canal
        A.salida --> delay propagacion datarate vtx --> B.entrada;
        display: "b=250,250";
endmodule

network Network : Protocolo1 // Network: Instancia del módulo Protocolo1
endnetwork
```

Escenario Protocolo1 Configuración del escenario para simular el protocolo "*simplex* sin restricciones".

Como se mencionó en el Capítulo 1, el escenario de simulación es descrito mediante lenguaje NED. El escenario de simulación consiste en definir que módulos componen el sistema y como se relacionan entre sí.

Para configurar el escenario de simulación del protocolo "*simplex* sin restricciones" se han definido dos módulos simples: módulo *Protocolo1Sender* y *Protocolo1Receiver* cuyos nombres deben coincidir con los nombres de las clases que los implementan, es decir, con las clases *Protocolo1Sender* y *Protocolo1Receiver*. Además, debido a que en el protocolo existe la transmisión unidireccional de datos, tanto el módulo *Protocolo1Sender* como el módulo *Protocolo1Receiver* presentan una sola compuerta (ver la Figura 2.7).

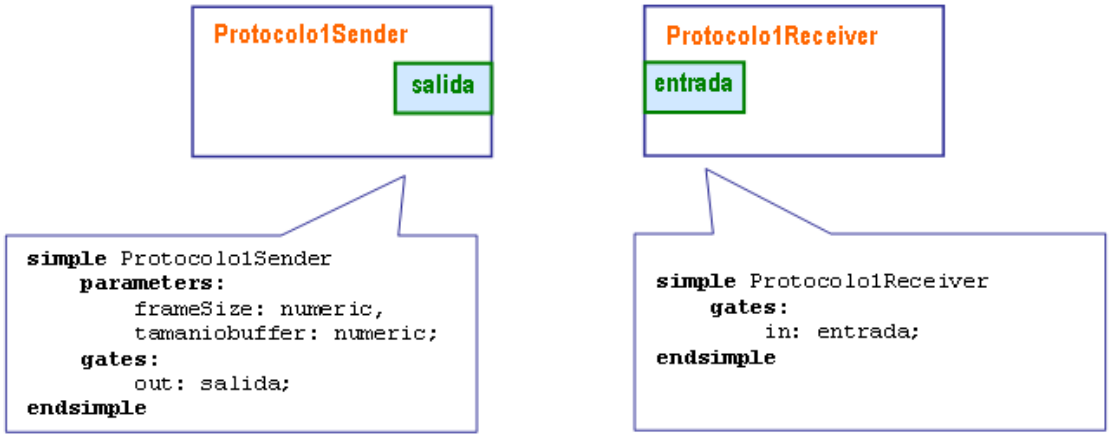


Figura 2.7 Definición de los módulos *Protocolo1Sender* y *Protocolo1Receiver.*

Posteriormente, con la sentencia *module Protocolo1*, se ha definido un módulo compuesto de nombre *Protocolo1*, dentro del cual se definen los parámetros:

- **propagacion:** Representa el tiempo de propagación en el canal.
- **vtx**: Representa la velocidad de transmisión en el canal.

Adicionalmente, se definen los parámetros *frameSize* y *npacket* cuyos valores a su vez serán asignados a los parámetros de los módulos simples.

Con la sentencia *submodules se* declaran los módulos simples que componen al módulo Protocolo1, que son: A, de tipo *Protocolo1Sender* y B, de tipo *Protocolo1Receiver*.

La sentencia *parameter* dentro de la declaración de un submódulo es utilizada para otorgarle un valor a los parámetros del submódulo. En este caso los valores se han hecho corresponder a los que tengan los parámetros del módulo compuesto.

Dentro del módulo compuesto se utiliza la sentencia *connections* para establecer las conexiones entre las compuertas de los submódulos y de esta manera formar el canal de comunicación; en este caso se ha conectado la compuerta *salida* del módulo *Protocolo1Sender* con la compuerta *entrada* del módulo *Protocolo1Receiver*. Además se asigna el valor de *propagacion* a *delay*, y *vtx* a *datarate*, con lo que se configura el tiempo de propagación y la velocidad de transmisión en el canal. En la Figura 2.8 se presenta el módulo y los submódulos configurados.

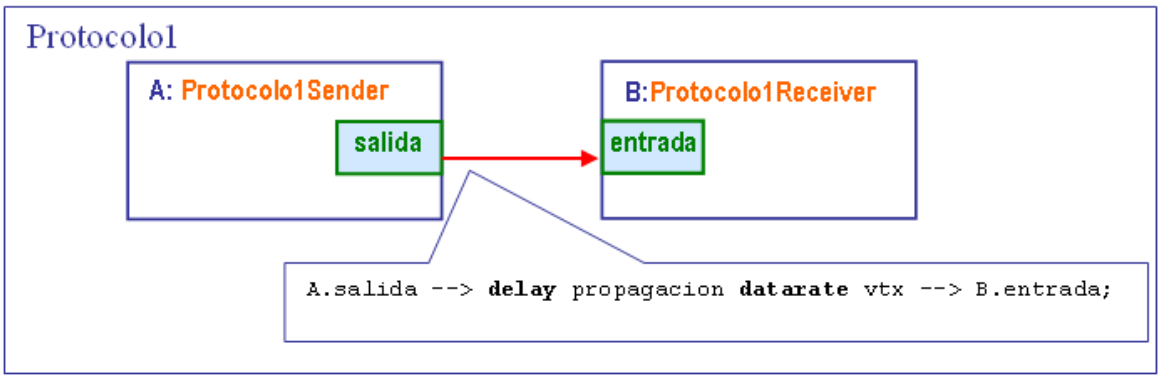


Figura 2.8 Definición del módulo compuesto *Protocolo1*.

Finalmente, se define la red de nombre *Network*, la cual es una instancia del módulo compuesto *Protocolo1*.

A continuación, en la Figura 2.9 se presenta el escenario configurado en la aplicación GNED para el protocolo "*simplex* sin restricciones."

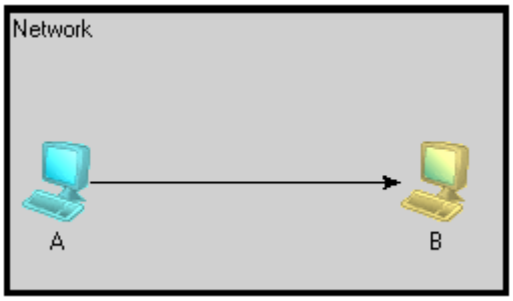


Figura 2.9 Escenario de simulación para el protocolo "*simplex* sin restricciones" presentado en GNED.

1.4.1.3.2. Configuración de los parámetros en el archivo omnetpp.ini

```
[General]
preload-ned-files=protocolo1.ned
network=Network
sim-time-limit = 1s

[Parameters]
Network.propagacion=0.050  # Tiempo de propagación en el canal en s
Network.vtx=10000000       # Velocidad de transmisión en bits por s
Network.frameSize=8000     # Tamaño de la trama en bits
Network.npacket=10         # Número de paquete que se obtendrán de la
                           # capa de red

[Run 0]
description="Protocolo1 con límite 10"

[Run 1]
description="Protocolo1 con límite 20"
Network.npacket=20
```

Figura 2.10 Configuración de los parámetros en omnetpp.ini para el protocolo "*simplex* sin restricciones"*.*

El archivo omnetpp.ini presentado en la Figura 2.10 es el archivo de configuración por defecto para el programa de simulación, este es leído en el momento que empieza la ejecución; contiene varias secciones, dentro de las que se configuran tanto los parámetros generales para la simulación como los parámetros de los módulos que intervienen en el escenario de simulación.

Las secciones de las que está compuesto el archivo son:

- **[General]:** Dentro de esta sección se definen los parámetros generales para la simulación, es decir aquellos que se aplican tanto para la configuración en modo gráfico *Tkenv* como en modo consola *Cmdenv*.

  En la primera sentencia (*preload-ned-files*) se especifica la ruta en donde se encuentran todos los módulos NED; en la siguiente sentencia (*network=Network*) se especifica la red a simularse, que en este caso es *Network*.

  A continuación con la sentencia *sim-time-limit=1s*, se especifica el tiempo en el que se terminará la simulación (1 segundo).

- **[Parameters]:** En esta sección se asigna los valores a los parámetros que no tienen asignado ninguno valor en los archivos NED, para este

caso se han asignado los valores para los parámetros de la red *Network* que son: *propagacion, vtx, frameSize, npacket*. En este archivo no es necesario asignar valores a los parámetros de los módulos simples *Protocolo1Sender* y *Protocolo1Receiver* ya que los valores de los parámetros de éstos son tomados desde el módulo compuesto.

- **[Run x]:** En esta sección se describen las diferentes opciones de simulación, las cuales podrán ser elegidas por el usuario. Además, en esta sección también se podrán configurar los parámetros para la simulación. En éste caso se presenta dos opciones: la primera opción se describe como "Protocolo1 con límite 10", en la que se tomará los valores por defecto para la red, y la segunda opción se describe como "Protocolo1 con límite 20" en la que se asigna a la variable *npacket* un valor de 20.

1.4.1.4. Compilación de los archivos .msg y .ned

Antes de compilar el proyecto en donde se ha implementado el nuevo protocolo es necesario compilar los archivos frame.msg y protocolo1.ned para generar los archivos .h y .cpp ya que de estos depende su compilación

Para la compilación del archivo frame.msg se realizan los siguientes pasos:

1. Se debe agregar el archivo frame.msg 

dentro del explorador de soluciones de Visual Studio 2003 para el proyecto sobre el cual se está implementando (ptop), como se presenta en la Figura 2.11.

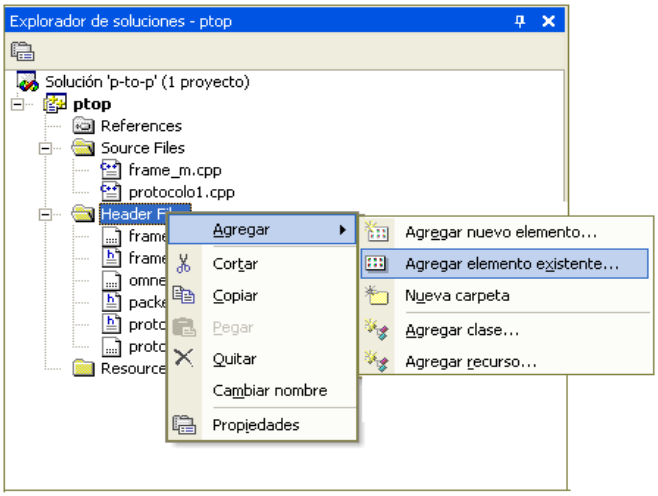


Figura 2.11 Agregar el archivo frame.msg al proyecto ptop.

2. Sobre el archivo añadido, se hace click derecho y se selecciona la opción “Propiedades”, a continuación la opción “Paso de generación personalizada”, y finalmente la opción “General” donde se deben agregar las líneas que permiten la compilación como se observa en la Figura 2.12.

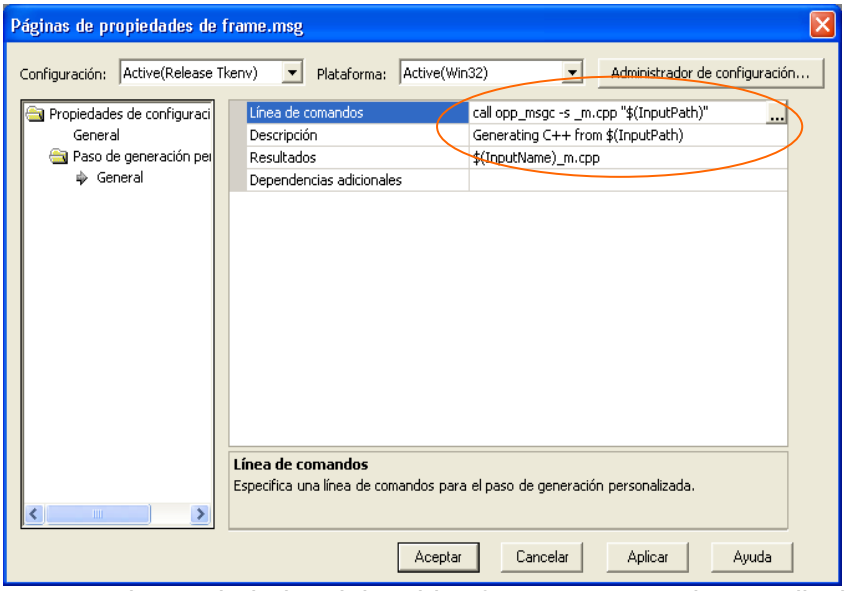


Figura 2.12 Ventana de propiedades del archivo frame.msg para la compilación en Visual Studio.

3. Se procede a la compilación, a partir de lo cual se crearán los archivos frame_m.h y frame_m.cpp que anteriormente fueron explicados, (ver la Figura 2.13).

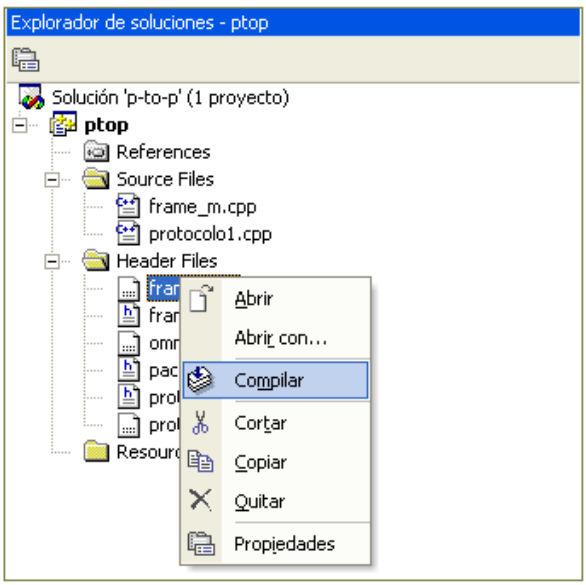
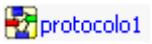


Figura 2.13 Compilación del archivo frame.msg.

Para la compilación del archivo protocolo1.ned se realizan los siguientes pasos:

1. Se debe agregar el archivo protocolo1.ned 

al proyecto ptop de la misma manera como se hizo con el archivo frame.msg.
2. Sobre el archivo añadido, se hace click derecho y se selecciona la opción “Propiedades”, a continuación la opción “Paso de generación personalizada”, y finalmente la opción “General” donde se deben agregar las líneas que permiten la compilación, como se observa en la Figura 2.14.

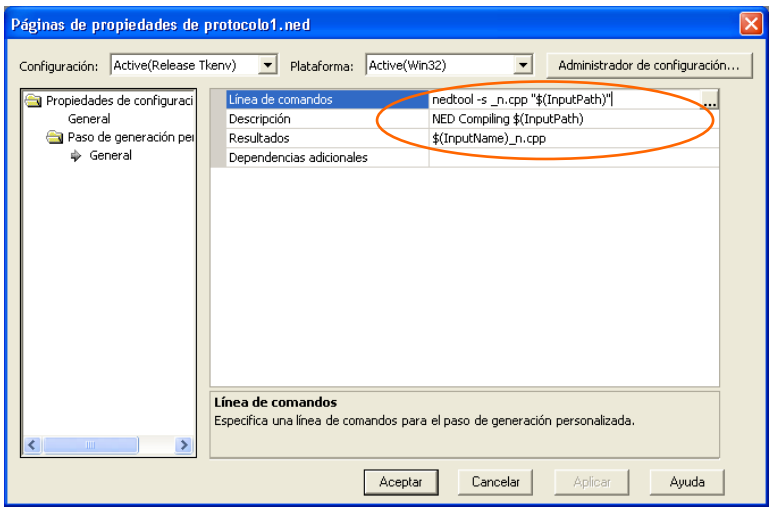


Figura 2.14 Ventana de propiedades del archivo protocolo1.ned para la compilación en Visual Studio.

3. Se procede a la compilación, a partir de lo cual se creará el archivo *protocolo1_n.cpp*.

El código generado en el archivo *protocolo1_n.cpp* básicamente permite relacionar todo lo que se ha definido en el archivo .ned con las clases correspondientes en C++; por ejemplo, se genera una clase derivada de la clase *cCompoundModule* de nombre *Protocolo1* para el módulo compuesto *Protocolo1* definido en el archivo .ned.

1.4.1.5. Ejecución de la simulación

Una vez compilado el proyecto ptop, se obtendrá el archivo protocolo1.exe el cual se ejecutará para iniciar la simulación.

El escenario de simulación se ejecutará con la interfaz gráfica, en donde se presentarán de forma animada todos los mensajes que se intercambian entre los módulos durante la simulación.

En la Figura 2.15 se presenta la primera ventana que permite seleccionar las dos opciones configuradas en el archivo omnetpp.ini para iniciar la simulación.

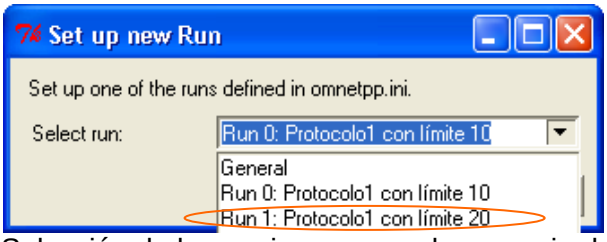


Figura 2.15 Selección de las opciones para el escenario de simulación.

Luego de haber seleccionado una de las opciones de la ventana anterior, el *kernel* del simulador, en este caso invocará a la función *initialize()* del módulo *Protocolo1Sender,* la misma que enviará inmediatamente (al tiempo 0s) 10 tramas al módulo *Protocolo1Receiver*. En la Figura 2.16 se presenta la ventana principal de la simulación.

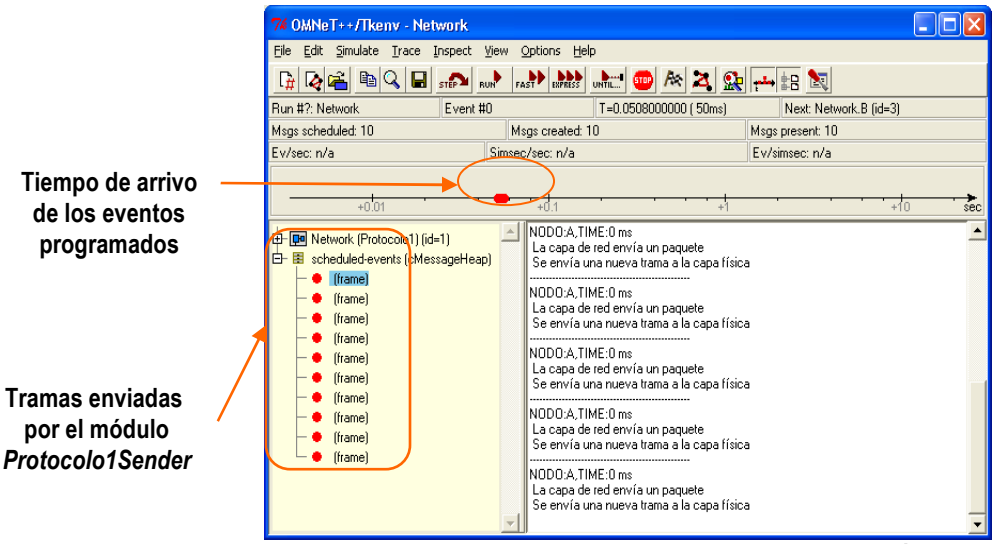


Figura 2.16 Ventana principal de la simulación.

Como se explicó en la Sección 1.2.4, la ventana principal presenta varios controles para ejecutar la simulación. En ella también se puede observar la planificación y ejecución de los eventos (ver la Figura 2.17).

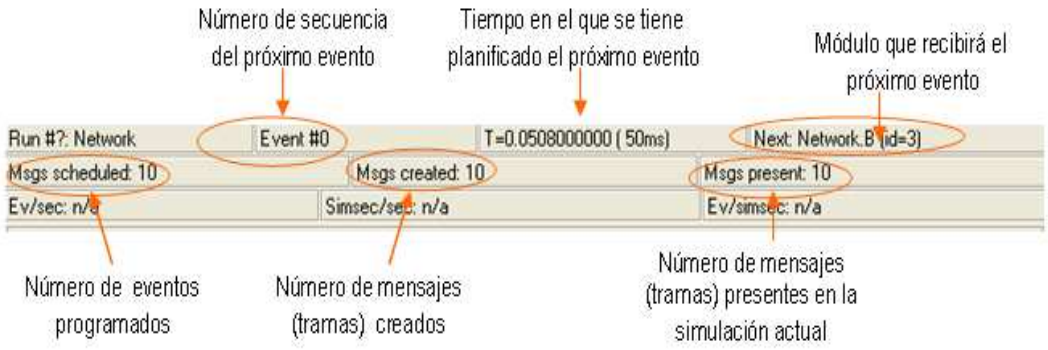
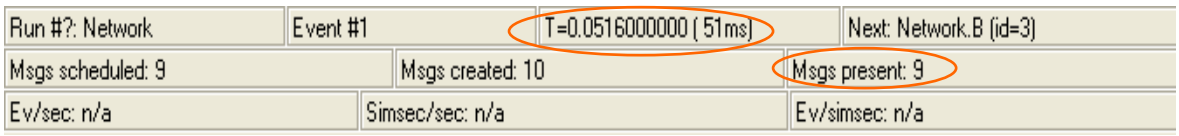


Figura 2.17 Presentación de eventos en la ventana principal de la simulación.

A medida que se ejecuta la simulación, se visualizarán cambios en la presentación de eventos en la ventana principal de la simulación como se muestra en la Figura 2.18.



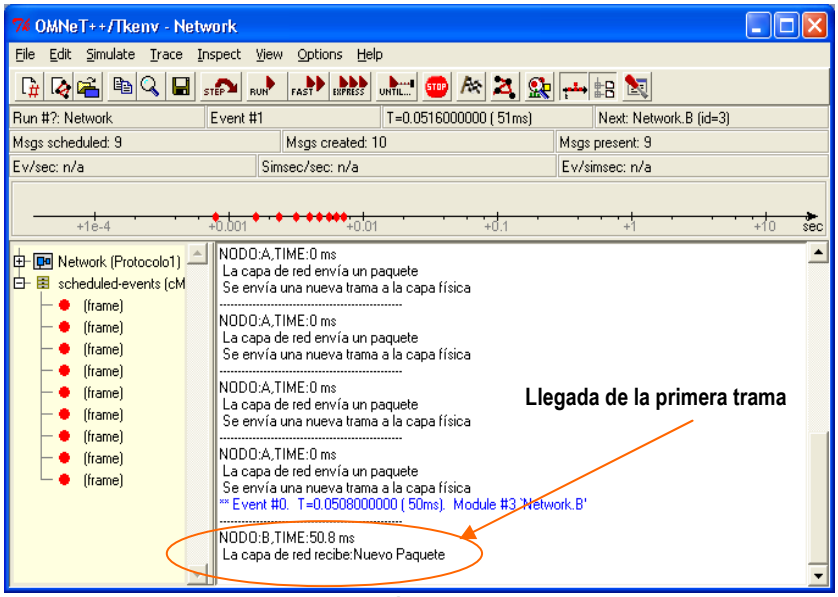


Figura 2.18 Cambios en la presentación de eventos en la ventana principal de la simulación luego de haberse consumido el primer evento.

En la Figura 2.19 se presentan, de forma gráfica, los componentes del escenario de simulación del protocolo "*simplex* sin restricciones"

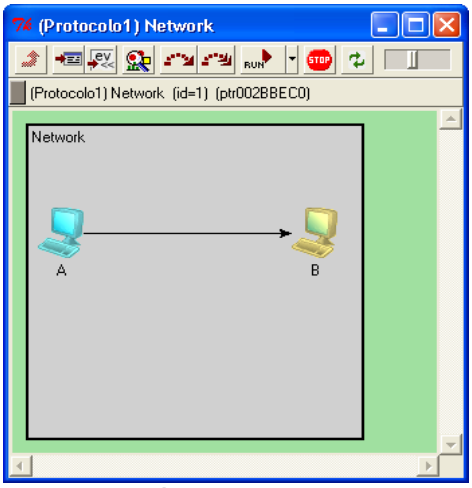


Figura 2.19 Escenario de simulación para el protocolo "*simplex* sin restricciones".

Además, se podrá ver la estructura interna de los módulos y su estado en tiempo de ejecución; ésto se obtiene de la opción *"Show object tree"* de la ventana principal, una vez realizado lo descrito antes, hacer click derecho en *"gates"* del módulo que se seleccione, nuevamente click derecho lo que permitirá escoger *"Inspect As Object"* y finalmente click derecho para escoger *"Inspec As Graphic".* En la Figura 2.20 se presenta la estructura interna de los módulos.

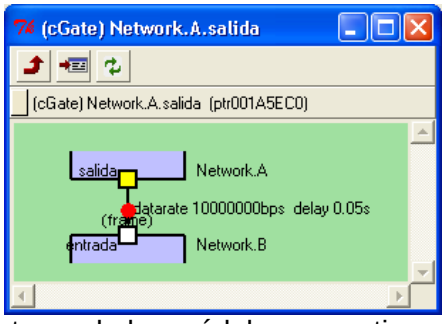


Figura 2.20 Estructura interna de los módulos en un tiempo dado de la simulación.

1.4.1.6. Resultados de la simulación

Debido a que se ha configurado una velocidad de transmisión de 10 Mbps y un tamaño de trama de 8000 bits se obtiene un tiempo de transmisión de 0,8 ms por trama, el tiempo de propagación en el canal es de 50ms, de manera que el tiempo total que se tarda la primera trama en llegar al módulo receptor es de 50,8ms.

Las tramas posteriores serán recibidas por el módulo B conforme se vayan desencolando, es así, por ejemplo, que la segunda trama es recibida al tiempo 51,6ms (0,8ms. más tarde desde que la primera trama fue recibida).

1.4.1.6.1. Impresión en pantalla

A continuación, en la Figura 2.21 se indican los resultados que se imprimen en la pantalla al realizar la simulación del protocolo "*simplex* sin restriciones".

```
-------------------------------------------------
NODO:A,TIME:0 ms
 La capa de red envía un paquete
 Envía una nueva trama a la capa física
-------------------------------------------------
NODO:A,TIME:0 ms
 La capa de red envía un paquete
 Envía una nueva trama a la capa física
-------------------------------------------------
NODO:A,TIME:0 ms
 La capa de red envía un paquete
 Envía una nueva trama a la capa física
.
.
.

                                        Generado por el simulador
** Event #0.  T=0.0508000000 ( 50ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B,TIME:50.8 ms
 La capa de red recibe:Nuevo Paquete
** Event #1.  T=0.0516000000 ( 51ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B,TIME:51.6 ms
 La capa de red recibe:Nuevo Paquete
** Event #2.  T=0.0524000000 ( 52ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B,TIME:52.4 ms
 La capa de red recibe:Nuevo Paquete
** Event #3.  T=0.0532000000 ( 53ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B,TIME:53.2 ms
 La capa de red recibe:Nuevo Paquete
** Event #4.  T=0.0540000000 ( 54ms).  Module #3 `Network.B'
-------------------------------------------------
.
.
```

Figura 2.21 Impresión en pantalla obtenida al simular el protocolo "*simplex* sin restricciones".

En los resultados presentado en la Figura 2.21 se puede observar que las tramas llegan al módulo receptor de acuerdo al tiempo calculado (tiempo de propagación + tiempo de transmisión), por ejemplo la primera trama llega a tiempo 50.8ms y cada 0.8ms después llegan las siguientes tramas,

1.4.1.6.2. Visualización gráfica de la simulación

En la Figura 2.22 se presenta la llegada de las tramas enviadas desde el módulo emisor hacia del módulo receptor en la que, debido a que todas las tramas se las envía al tiempo 0s, se observa sólo la llegada de la última trama.

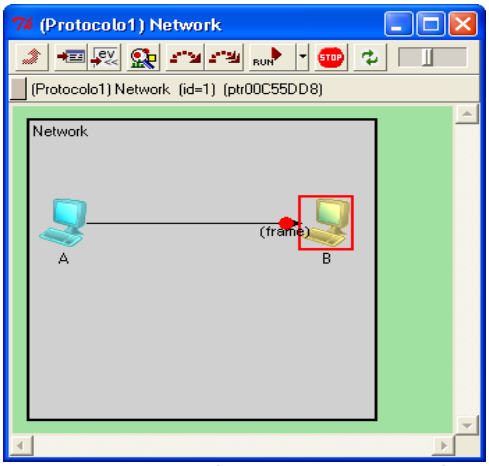


Figura 2.22 Llegada de la última trama al módulo receptor.

De los resultados obtenidos se puede concluir que el protocolo cumple con las características descritas para el mismo, es así que el módulo emisor envía tramas sin ninguna limitación, y el módulo receptor las recibe de igual forma.

1.4.2. PROTOCOLO SIMPLEX DE PARADA Y ESPERA

1.4.2.1. Descripción del protocolo

Adicionalmente a las funciones de la capa de enlace de datos implementadas en el protocolo descrito en la Sección 1.4.1, en éste se incluye la función de control de flujo (ver el Anexo C).

Las consideraciones para este protocolo son las siguientes **[7]**:

- Los datos se transmiten sólo en una dirección.
- El canal de comunicación nunca tiene problemas, ni pierde tramas.
- La capacidad del búfer del receptor es finita.
- La capacidad de procesamiento del receptor es finita.

Las dos últimas consideraciones citadas pueden causar el siguiente problema: el emisor puede saturar al receptor enviando datos a mayor velocidad de la que este último puede procesarlos.

1.4.2.2. Diseño e implementación en C++

El problema que surge por la limitación de búfer y de procesamiento, se ha solucionado con la implementación de control de flujo, en el que el receptor debe enviar una trama de confirmación (acuse de recibo) que informe al emisor que puede continuar con la transmisión de datos.

A continuación, la Figura 2.23 presenta el diagrama de flujo para el módulo emisor, y la Figura 2.24 presenta el diagrama de flujo par al módulo receptor.

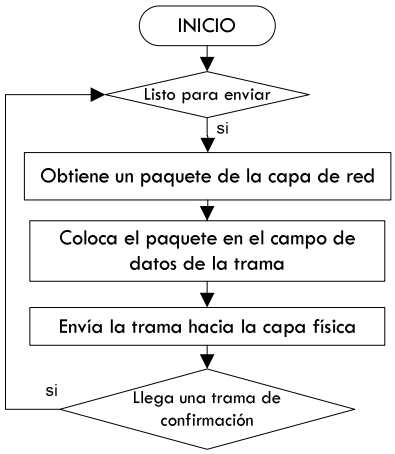


Figura 2.23 Diagrama de flujo para el módulo emisor.

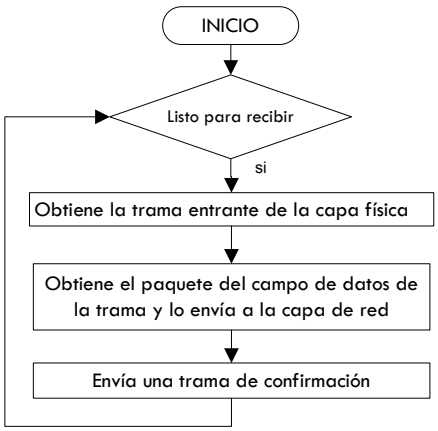


Figura 2.24 Diagrama de flujo para el módulo receptor.

Tomando en cuenta la consideración de transmisión unidireccional de datos, se requiere implementar la clase *Protocolo2Sender* que represente al emisor y la clase *Protocolo2Receiver* que represente al receptor.

1.4.2.2.1. Módulos

Clase Protocolo2Sender

El objetivo de esta clase es enviar la trama de datos al receptor y esperar por una trama de confirmación de éste para continuar con la transmisión de una nueva trama de datos, esto se lo implementa mediante las funciones: *initialize()* y *handleMessage().*

La definición de la clase *Protocolo2Sender* se presenta en el Código Protocolo 2.1.

```
class Protocolo2Sender:public cSimpleModule
{
private:
     frame *s;          // Variable que representa a la trama
                        // que será enviada
     packet buffer;     // Paquete que se obtiene de la
                        // capa de red
     int frameSize;     // Tamaño de la trama

public:
// Constructor
     Protocolo2Sender();

// Destructor
     virtual ~Protocolo2Sender();

// Funciones del protocolo
     void from_network_layer(packet*  msg);    // Obtiene un paquete
                                               // de la capa de red
     void to_physical_layer(frame *psend);     // Envía una trama
                                               // hacia la capa física
     void crear_paquete(packet *msg);          // Genera un paquete en la
                                               // capa de red
     // Función auxiliar
     void registro_evento(); // Imprime en pantalla el nombre del nodo
                             // y el tiempo en que ocurre un
                             // evento
protected:
// Funciones de la clase cSimpleModule que se redefinen
     virtual void initialize();    // Función que es invocada para
                                   // iniciar la simulación
     virtual void handleMessage(cMessage *msg)  // Función que recibe
                                                // la trama desde la
                                                // capa física

};
```

Código Protocolo 2.1 Definición de la clase *Protocolo2Sender***.**

Funciones

- **initialize():** Función en la que se envía la primera trama de datos, es decir empezará con la transmisión, debido a que esta función es invocada por el *kernel* para iniciar la simulación, en ésta se hace la planificación del primer evento. En el Código Protocolo 2.2 se presenta la implementación de la función.

```
void Protocolo2Sender::initialize()
{
     // Se imprime en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();

     // Se toma el valor configurado en el archivo de
     // inicialización .ini o en el archivo .ned
     frameSize= par("frameSize");

     // Se reserva espacio de memoria para la
     // trama que se enviará
     s=new frame(“Datos”);

     // Se obtiene un paquete de la capa de red
     from_network_layer(&buffer);

     // Se coloca el paquete en el campo info de la trama
     s->setInfo(buffer);

     // Se asigna el tamaño de la trama de acuerdo al valor
     // configurado en el archivo .ini o .ned
     s->setLength(frameSize);

     // Se envía la trama a la capa física
     to_physical_layer(s);

     ev<<" ...Esperando la confirmación de recepción"<<endl;

}
```

Código Protocolo 2.2 Implementación de la función *initialize()* de la clase *Protocolo2Sender*.

- ***handleMessage():*** Esta función se ha implementado considerando que es necesario recibir una confirmación por parte del receptor para continuar con la transmisión; una vez que se reciba la trama de confirmación, la capa de enlace de datos podrá recibir un nuevo paquete de la capa de red, realizar la función de entramado y enviar la trama a la capa física, finalmente, se borra la trama de confirmación que fue enviada por el receptor (esto ocurre debido a que desde el punto de vista de C++

el objeto es el mismo que se creó en el módulo receptor). En el Código Protocolo 2.3 se presenta la implementación de la función.

```
void Protocolo2Sender::handleMessage(cMessage *p_r_)
{
     // Función que imprimirá en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();

     ev<<" Recibe la trama de confirmación"<<endl;

     // Se reserva espacio de memoria para la
     // trama que se enviará
     s=new frame(“Datos”);

     // Se obtiene un paquete de la capa de red
     from_network_layer(&buffer);

     // Se coloca el paquete en el campo info de la trama
     s->setInfo(buffer);

     // Se asigna el tamaño de la trama de acuerdo al valor
     // configurado en el archivo .ini o .ned
     s->setLength(frameSize);

     // Se envía la trama a la  capa física
     to_physical_layer(s);

     // Borra la trama de confirmación recibida
     delete p_r_;
     ev<<" ...Esperando la confirmación de recepción"<<endl;

}
```

Código Protocolo 2.3 Implementación de la función *handleMessage()* de la clase *Protocolo2Sender*.

El cuerpo de la función *handleMessage()* e *initialize()* son similares, la diferencia entre estas funciones radica en que *handleMessage()* es invocada cada vez que llega una trama al módulo, mientras que *initialize()* es invocada sólo para iniciar la transmisión.

Las funciones restantes son similares a las discutidas en el protocolo “*simplex* sin restricciones”.

Clase Protocolo2Receiver

El objetivo de esta clase es recibir la trama de datos, extraer el paquete del campo de datos de la trama, enviarlo a la capa de red, y posteriormente enviar una nueva trama de confirmación al emisor; todo este proceso se lo realiza en la función *handleMessage().*

La definición de la clase *Protocolo2Receiver* es presentada en el Código Protocolo 2.4.

```
class Protocolo2Receiver:public cSimpleModule
{

private:
     frame *s;         // Variable que representa a la trama de
                       // confirmación que enviará el receptor

     int frameSize;    // Tamaño de la trama

public:
// Constructor
     Protocolo2Receiver();

// Destructor
     virtual ~Protocolo2Receiver();

// Funciones del protocolo
     void to_network_layer(packet *msg);      // Envía un paquete a
                                              // la capa de red

     void to_physical_layer(frame *psend);    // Envía una trama
                                              // hacia la capa física
// Función auxiliar
     void registro_evento(); // Imprime en pantalla el nombre del nodo
                             // y el tiempo en que ocurre un
                             // evento
protected:
// Función de la clase cSimpleModule que se redefine
     virtual void handleMessage(cMessage *msg);// Función que recibe
                                               // la trama desde la
                                               // capa física
};
```

Código Protocolo 2.4 Definición de la clase *Protocolo2Receiver*.

Variables

- **s:** Variable que representa a la trama de confirmación que se enviará al emisor luego de haber recibido una trama de datos.

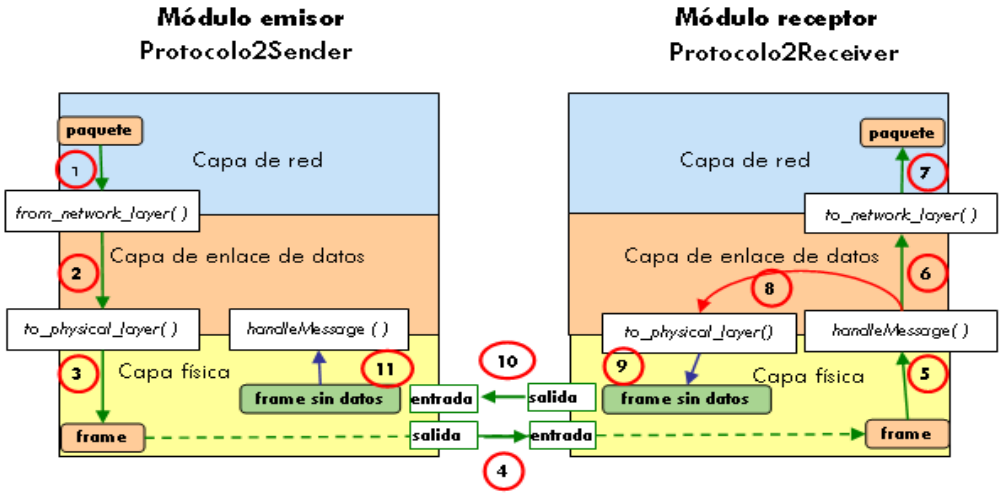
Funciones

- ***handleMessage*():** Función en la que luego de recibir la trama de datos y procesarla, envía una trama de confirmación para cumplir con la función de retroalimentación. Su implementación es presentada en el Código Protocolo 2.5.

```
void Protocolo2Receiver::handleMessage(cMessage *p_r_)
{
     // Se imprime en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();
     // Convierte el objeto recibido de tipo cMessage a tipo frame
     frame *r = check_and_cast<frame*>(p_r_);
     ev<<" Se recibe la trama de datos"<<endl;
     // Se envían los datos hacia la capa de red
     to_network_layer(&r->getInfo());
     // Se crea una nueva trama para enviar como confirmación
     s=new frame("Confirmación");
     // Se asigna el tamaño a la trama de confirmación
     s->setLength(80);
     // Se envía la trama a la capa física
     to_physical_layer(s);
     // Se borra la trama recibida
     delete p_r_;
}
```

Código Protocolo 2.5 Implementación de la función *handleMessage()* de la clase *Protocolo2Receiver*.

A continuación, en la Figura 2.25 se presenta gráficamente el proceso que desencadenan los módulos emisor y receptor implementados para el protocolo "*simplex* de parada y espera".



1.- La función *from_network_layer( )* recibe un paquete de la capa de red.
2.- El paquete recibido se encapsula en una trama.
3.- La función *to_physical_layer()* envía la trama creada hacia la capa física.
4.- Se envía la trama desde el módulo emisor hacia el módulo receptor
5.- La función *handleMessage()* del módulo receptor recibe la trama.
6.- Se desencapsula la trama y se obtiene el paquete.
7.- La función *to_network_layer()* envía el paquete a la capa de red.
8.- Se crea una trama de confirmación que se enviará de retorno al módulo emisor.
9.- La función *to_physical_layer()* envía la trama de confirmación hacia la capa física.
10.- Se envía la trama de confirmación desde el módulo receptor hacia el módulo emisor.

Posteriormente el emisor enviará un nuevo paquete y se repetirá el proceso antes detallado

Figura 2.25 Proceso que sigue el protocolo *simplex* de parada y espera.

1.4.2.3. Configuración del escenario de simulación

1.4.2.3.1. Configuración del escenario de simulación en el lenguaje NED

```
// Definición del módulo emisor
simple Protocolo2Sender
    parameters:
        frameSize: numeric; // Tamaño de la trama

       gates:
        in: entrada; // Compuerta de entrada
        out: salida; // Compuerta de salida
endsimple
// Definición del módulo receptor
simple Protocolo2Receiver
    gates:
        in: entrada; // Compuerta de entrada
        out: salida; // Compuerta de salida
endsimple

// Definición de un módulo compuesto formado de
// los módulos simples emisor y receptor
module Protocolo2
    parameters:
        frameSize: numeric,
        propagacion: numeric, // Tiempo de propagación
        vtx: numeric;         // Velocidad de transmisión

    submodules:
        A: Protocolo2Sender;  // Instancia del módulo Protocolo2Sender
            parameters:
                frameSize = frameSize;
            display: "p=35,96;i=device/pc,cyan";
        B: Protocolo2Receiver;// Instancia del módulo Protocolo2Receiver
            display: "p=224,96;i=device/pc,gold";
    connections:
        // Configuración de los parámetros del canal
        A.salida --> delay propagacion datarate vtx --> B.entrada;
        B.salida --> delay propagacion datarate vtx --> A.entrada;

    display: "b=250,250";
endmodule

network Network : Protocolo2 // Network: Instancia del módulo Protocolo2
endnetwork
```

Escenario Protocolo2 Configuración del escenario para simular el protocolo "*simplex* de parada y espera".

Cabe indicar que aunque la transmisión es unidireccional, se definen dos enlaces y la comunicación es en ambos sentidos, ya que el receptor tendrá que enviar la trama de confirmación ante la llegada de una trama de datos, por tanto en cada módulo es necesario definir tanto una compuerta de entrada como una de salida, para posteriormente conectarlas como se presentan en la Figura 2.26.

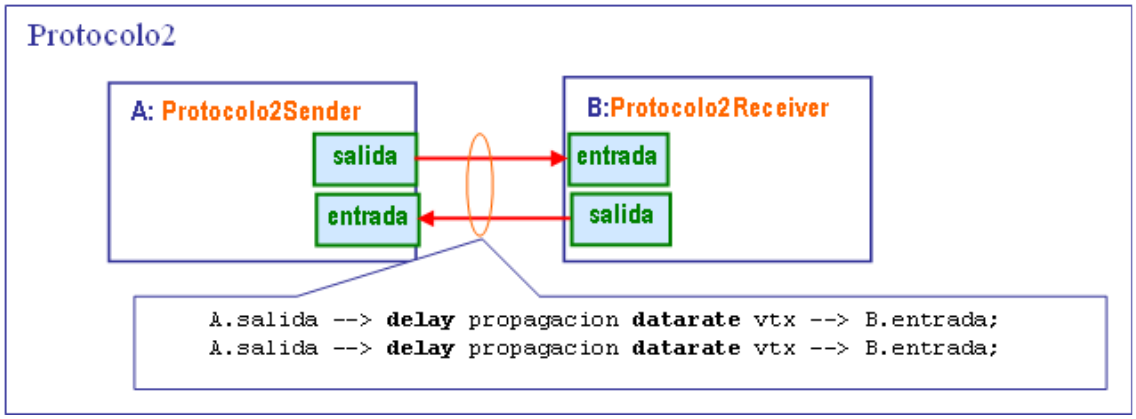


Figura 2.26 Definición del módulo compuesto *Protocolo2*.

A continuación, en la Figura 2.27 se presenta el escenario configurado en la aplicación GNED para el protocolo "*simplex* con parada y espera."

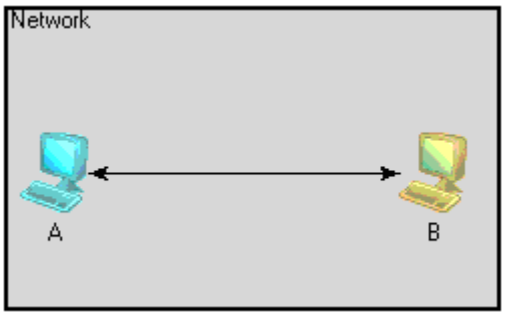


Figura 2.27 Escenario de simulación para el protocolo "*simplex* de parada y espera" presentado en GNED.

1.4.2.3.2. Configuración de los parámetros en el archivo omnetpp.ini

```
[General]
preload-ned-files=protocolo2.ned
network=Network
sim-time-limit = 0.1s

[Parameters]
Network.propagacion=0.050    # Tiempo de propagación en el canal en s
Network.vtx=10000000         # Velocidad de transmisión en bits por s

Network.frameSize=8000       # Tamaño de la trama en bits

[Run 0]
description="Protocolo 2"
```

Figura 2.28 Configuración de los parámetros en omnetpp.ini para el protocolo "*simplex* de parada y espera".

1.4.2.4. Resultados de la simulación

Tanto la velocidad de transmisión, el tamaño de la trama de datos, y el tiempo de propagación tienen los mismos valores que los configurados para el protocolo "*simplex* sin restricciones".

El tamaño asignado para la trama de confirmación es de 80 bits, por tanto el tiempo en que tardará en llegar al módulo emisor será 50,08ms.

1.4.2.4.1. Impresión en pantalla

A continuación, en la Figura 2.29 se indican los resultados que se imprimen en la pantalla al realizar la simulación del protocolo "*simplex* de parada y espera".

```
-------------------------------------------------
NODO:A,TIME:0 ms
 La capa de red envía un paquete
 Envía una nueva trama a la capa física
 ...Esperando la confirmación de recepción
** Event #0.  T=0.0508000000 ( 50ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B,TIME:50.8 ms
 Recibe la trama de datos
 La capa de red recibe:Nuevo Paquete
 Envía una trama de confirmación a la capa física
** Event #1.  T=0.1008080000 (100ms).  Module #2 `Network.A'
-------------------------------------------------
NODO:A,TIME:100.808 ms
 Recibe la trama de confirmación
 La capa de red envía un paquete
 Envía una nueva trama a la capa física
 ...Esperando la confirmación de recepción
** Event #2.  T=0.1516080000 (151ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B,TIME:151.608 ms
 Recibe la trama de datos
 La capa de red recibe:Nuevo Paquete
 Envía una trama de confirmación a la capa física
** Event #3.  T=0.2016160000 (201ms).  Module #2 `Network.A'
-------------------------------------------------
NODO:A,TIME:201.616 ms
 Recibe la trama de confirmación
 La capa de red envía un paquete
 Envía una nueva trama a la capa física
 ...Esperando la confirmación de recepción
```

Figura 2.29 Impresión en pantalla obtenida de la simulación del protocolo "*simplex* de parada y espera".

En la Figura 2.29 se observar que una vez que el módulo receptor recibe un trama éste envía una trama de confirmación al emisor (por ejemplo al tiempo 50.8ms el receptor recibe una trama de datos y envía una trama de confirmación), una vez

que el emisor recibe la trama de confirmación puede enviar una nueva trama de datos (por ejemplo al tiempo 100.808ms el emisor recibe la trama de confirmación y envía una nueva trama).

1.4.2.4.2. Visualización gráfica de la simulación

En la Figura 2.30 y Figura 2.31 se presenta la visualización grafica como resultado de la ejecución del escenario de simulación.

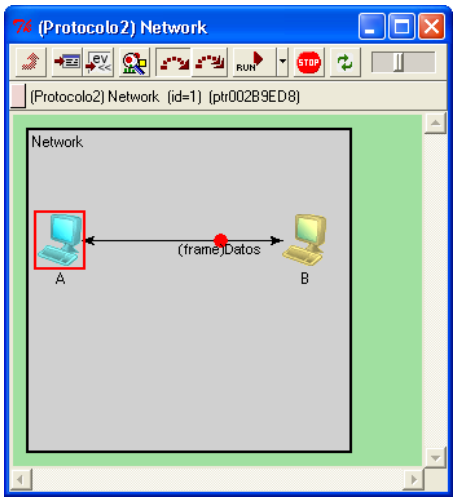


Figura 2.30 Envió de la trama de datos del módulo emisor hacia el módulo receptor.

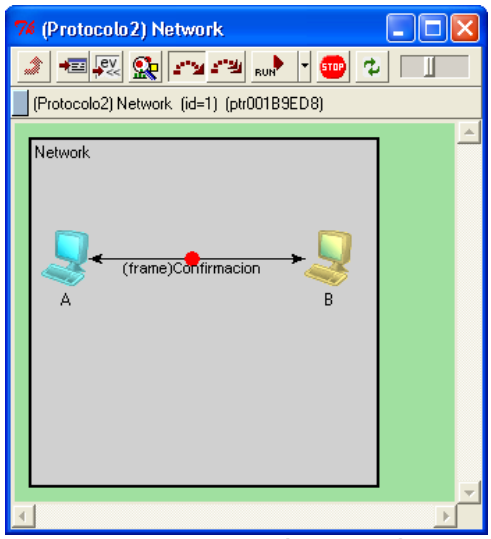


Figura 2.31 Envió de la trama de confirmación del módulo receptor hacia el módulo emisor.

De los resultados obtenidos se puede concluir que el protocolo cumple con la función de control de flujo que se consideró para su implementación, es asi que el emisor espera una trama de confirmación para poder enviar una nueva trama de datos.

1.4.3. PROTOCOLO SIMPLEX PARA UN CANAL CON RUIDO

1.4.3.1. Descripción del protocolo

A las consideraciones del protocolo anterior se incluyen:

- Un canal de comunicación que comete errores lo que se ve reflejado únicamente en la pérdida de tramas.

La presencia de un canal de comunicación que comete errores conlleva al problema que se explica mediante el siguiente escenario:

La capa de red de la máquina A pasa una serie de paquetes a la capa de enlace de datos, que debe asegurar que se entregue una serie de paquetes idénticos a la capa de red de la máquina B a través de su capa de enlace de datos. En particular, con los protocolos antes presentados, la capa de red de la máquina B no tiene manera de saber si el paquete se ha perdido o se ha duplicado, por lo que la capa de enlace de datos debe garantizar que ninguna combinación de errores de transmisión, por improbables que sean, pueda causar la entrega de un paquete duplicado a la capa de red. Para una descripción detallada del protocolo puede referirse al Anexo C.

1.4.3.2. Diseño e implementación en C++

Para solventar el problema mencionado en la descripción del protocolo "*simplex* para un canal con ruido", es necesario implementar un módulo receptor que sea capaz de distinguir entre tramas que está recibiendo por primera vez y una retransmisión. La forma evidente de lograr esto es hacer que el emisor ponga un número de secuencia en cada trama que envía, de tal manera que el receptor pueda examinar el número de secuencia de la trama que llega para determinar si es una trama nueva o una trama duplicada que debe descartarse.

Para este caso un número de secuencia de un bit (0 ó 1) será suficiente, ya que en cada instante el receptor espera un número de secuencia en particular. Cualquier trama de entrada que contenga un número de secuencia equivocado se rechaza como duplicado. Cuando llega una trama que contiene el número de secuencia correcto, se acepta, se obtiene el paquete y se lo pasa a la capa de

red, y el número de secuencia esperado cambia (si el valor es 0 se cambia a 1 y si el valor es 1 se cambia a 0).

En este protocolo se hace uso de un temporizador, el cual arranca luego que el emisor transmite una trama, si éste ya se estaba ejecuntado se reestablece para conceder otro intervalo completo de temporización. Solo cuando ha transcurrido el intervalo completo de temporización (ya sea porque se ha perdido la trama de datos o la confirmación de recepción) el emisor puede suponer con seguridad que se ha perdido la trama y debe enviar su duplicado.

A continuación, en la Figura 2.32 se presenta el diagrama de flujo para el módulo emisor y en la Figura 2.33 se presenta el diagrama de flujo para el módulo receptor.

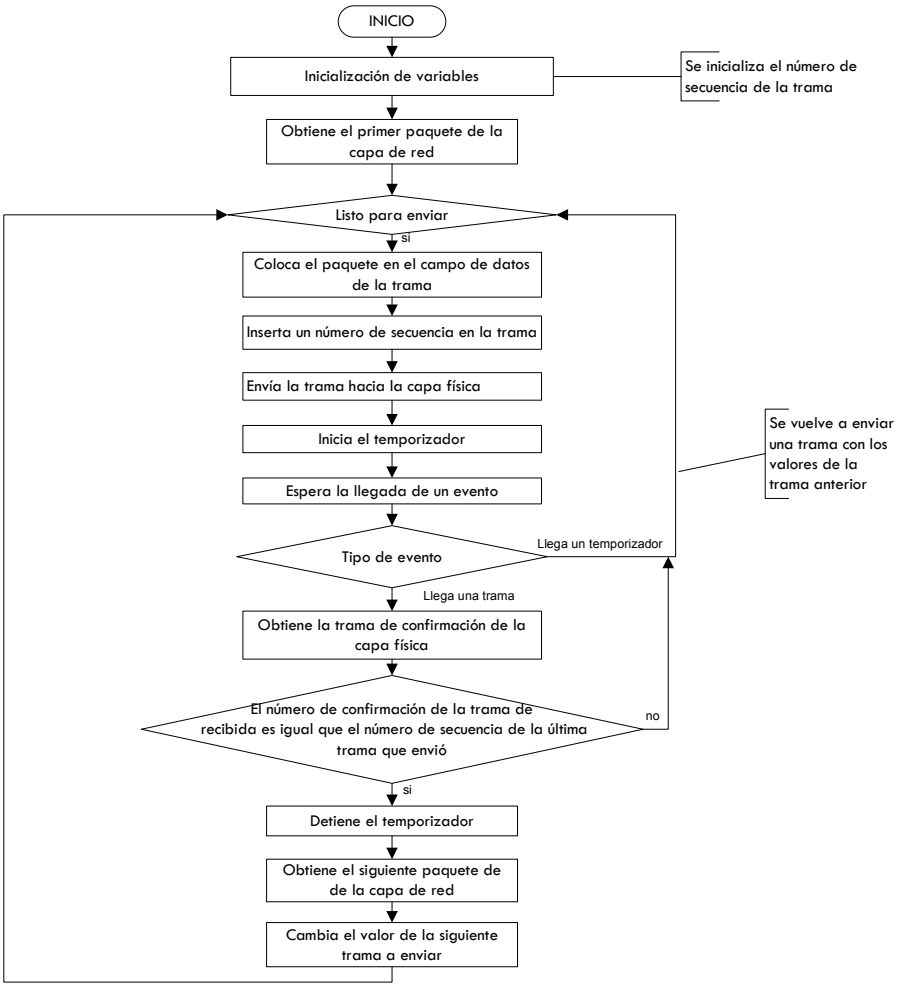


Figura 2.32 Diagrama de flujo del módulo emisor.

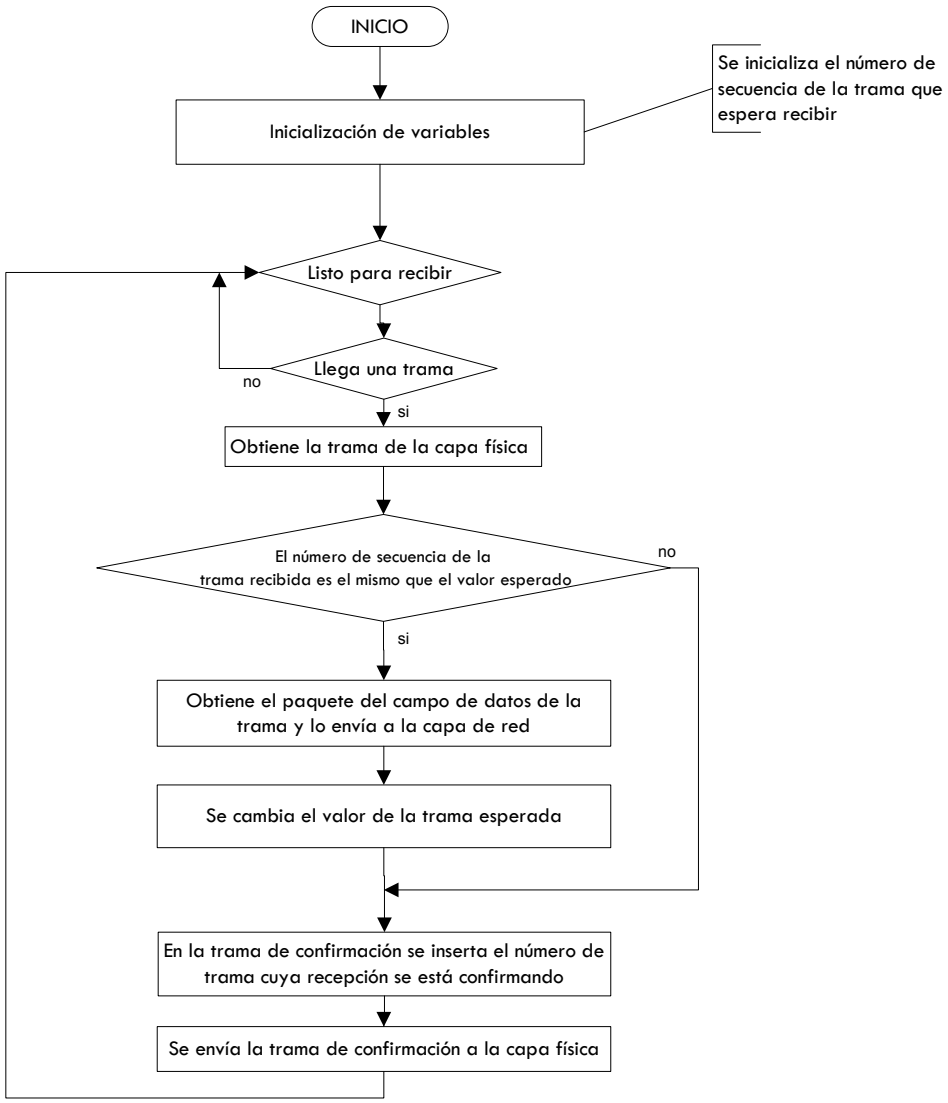


Figura 2.33 Diagrama de flujo del módulo receptor.

1.4.3.2.1. Temporizador

La implementación de un temporizador se basa en el envío de un objeto de la clase *cMessage* o de alguna clase derivada de ésta hacia el mismo módulo **[8]**.

La clase *cSimpleModule* provee funciones a través de las cuales se puede simular un temporizador, éstas son:

- **scheduleAt():** Función que toma dos argumentos, en el primero se indica el tiempo planificado (en segundos) en que llegará el objeto especificado en su segundo argumento; el objeto será recibido por la función *handleMessage()*, en donde se deberá implementar la funcionalidad para discriminar la llegada de un objeto que fue enviado por el mismo módulo (temporizador expirado)

de la llegada de una unidad de datos que fue enviada por un módulo diferente (trama).

- **cancelEvent():** Función que permite cancelar la llegada del objeto (planificada por la función *scheduleAt()*) antes que concluya el tiempo para el cual estaba programada.

Además de las funciones antes descritas, existen funciones de la clase *cMessage* que son utilizadas en la implementación de temporizadores, estas son:

- **isSelfMessage():** Función con la que se puede verificar si el objeto recibido fue enviado por el mismo módulo, si es así, la función devuelve *true* caso contrario devuelve *false*.
- **isScheduled():** Función que permite verificar si el objeto ya ha llegado a su destino.
- **arrivalTime():** Función que devuelve el tiempo en el que está planificada la llega del objeto.

En OMNET++, la clase *cModule* provee la función *bubble()* que permite la visualización durante la simulación del mensaje configurado en su argumento, esto facilitará al usuario estar al tanto de algunos eventos que pueden ocurrir **[9]**. En la Figura 2.34 se presenta la cadena de caracteres que se envió como argumento en la función *bubble()* .

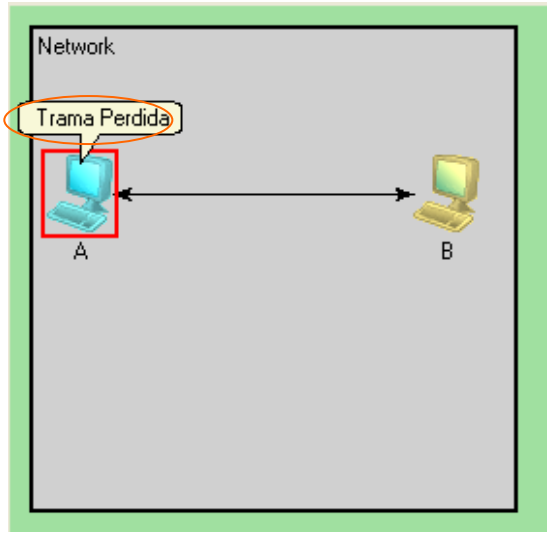


Figura 2.34 Visualización de la cadena de caracteres que se envió en el argumento de la función *bubble()*.

1.4.3.2.2. Módulos

Clase Protocolo3Sender

El objetivo de esta clase es transmitir una trama y arrancar un temporizador, posteriormente esperar a que ocurra uno de los siguientes eventos:

- Llegada de una trama de confirmación de recepción.
- Expiración del temporizador.

El Código Protocolo 3.1 presenta la definición de la clase *Protocolo3Sender.*

```
#define MAX_SEQ 1      // Máximo número que tendrá la secuencia
typedef int seq_nr;    // Número de secuencia o de confirmación

class Protocolo3Sender:public cSimpleModule
{
private:
// Variables de protocolo
     frame *s;                     // Variable que representa a la trama
                                   // que será enviada
     frame *r;                     // Variable que representa a la trama
                                   // que será recibida
     packet buffer;                // Paquete que se obtiene  de la
                                   // capa de red
     int frameSize;                // Tamaño de la trama
     double timer;                 // Variable que representa al
                                   // intervalo de tiempo que espera
                                   // antes que el temporizador expire
     seq_nr next_frame_to_send;    // Variable que representa
                                   // el número de secuencia de
                                   // la siguiente trama de salida
     double channel_behavior;      // Variable que representa el
                                   // comportamiento del canal a través
                                   // de una distribución de
                                   // probabilidad, en base a esta se
                                   // conoce cuando se generan
                                   // pérdidas de tramas
     double loss;                  // Variable que representa la tasa de
                                   // pérdida en el canal considerando
                                   // el valor de channel_behavior


     cMessage *protocolo_timer;    // Variable que representa
                                   // al temporizador


public:
// Constructor
     Protocolo3Sender();

// Destructor
     virtual ~Protocolo3Sender();
```

```
// Funciones del protocolo

    void from_network_layer(packet* buffer);  // Obtiene un paquete
                                              // de la capa de red
     void to_physical_layer(frame * envio);   // Envía una trama
                                              // hacia la capa física
// Funciones para el temporizador
     void start_timer(cMessage *timer);       // Inicia el temporizador
     void stop_timer();                       // Detiene el temporizador

// Funciones auxiliares
     void crear_paquete(packet *msg);    // Genera un paquete en la capa
                                         // de red
     void registro_evento();             // Imprime en pantalla el
                                        // nombre del nodo
                                         // y el tiempo en que ocurre un
                                         // evento
     void inc(int& k);                   // Incrementa circularmente

     void retransmision();               // Retransmite una trama cuando
                                         // expira el temporizador.

protected:

// Funciones de la clase cSimpleModule que se redefinen
     virtual void initialize();          // Función que es invocada para
                                         // iniciar la simulación
     virtual void handleMessage(cMessage *p_r_); // Función que recibe
                                                 // la trama desde la
                                                 // capa física o un
                                                 // objeto que
                                                 // representa a un
                                                 // temporizador
                                                 // expirado
};
```

Código Protocolo 3.1 Definición de la clase *Protocolo3Sender*.

Variables

- **next_frame_to_send:** Variable de tipo *int* (declarada como *seq_nr)* con la que el emisor recuerda el número de secuencia de la siguiente trama a enviar.
- **timer:** Variable que representa el intervalo de tiempo en el que el temporizador expirará. Dicho intervalo debe escogerse de modo que haya suficiente tiempo para que la trama llegue al receptor, sea procesada y retorne la confirmación de recepción al emisor.
- **protocolo_timer:** Variable que representa al temporizador.

- **channel_behavior:** Variable que representa el comportamiento del canal de comunicaciones mediante una función de distribución de probabilidad que OMNET++ provee y que es determinada en el archivo de inicialización .ini o en el archivo .ned, en base a ésta se generan las pérdidas de tramas en el canal.
- **loss:** Variable que representa la tasa de pérdida de tramas considerando el valor de channel_behavior; su valor es configurado en el archivo de inicialización .ini o en el archivo .ned.

Funciones

- **Protocolo3Sender():** Constructor en el que se inicializa el puntero de tipo *cMessage* que representa al temporizador como se muestra en el Código Protocolo 3.2.

```
Protocolo3Sender::Protocolo3Sender()
{
     // Se inicializa el puntero protocolo_timer
     // al cual se le da el nombre timerout
     protocolo_timer=new cMessage("timerout");

}
```

Código Protocolo 3.2 Implementación del constructor de la clase *Protocolo3Sender.*

- **~Protocolo3Sender():** Destructor en el que se hace uso de la función *CancelAndDelete(),* la cual es utilizada para liberar espacio de memoria asignado al mensaje que simula al temporizador, esta función inicialmente verifica que el mensaje no esté en la lista de eventos futuros, en caso de estarlo lo cancela para luego borrarlo. La implementación del destructor se presenta en el Código Protocolo 3.3.

```
Protocolo3Sender::~Protocolo3Sender()
{
     // Se libera el espacio de memoria asignado para
     // el temporizador pero antes se verifica si no
     // está en la lista de eventos futuros caso contrario
     // cancela el evento para proceder a borrarlo
     cancelAndDelete(protocolo_timer);
     // Se libera el espacio  de memoria asignado a
     // al puntero s que representa a la trama que se
     // enviará
     delete s;
}
```

Código Protocolo 3.3 Implementación del destructor de la clase *Protocolo3Sender.*

- **start_timer():** Función que reprograma el intervalo de tiempo (*simTime()+timer*) en el que expirará el temporizador *protocolo_timer*. Para esto se hace uso de la función *scheduleAt()* de la clase *cSimpleModule* anteriormente explicada. Su implementación se la presenta en Código Protocolo 3.4.

```
void Protocolo3Sender::start_timer(cMessage *event)
{
     ev<<name()<<" Iniciando el temporizador\n";
     // Se reprograma el temporizador
     scheduleAt(simTime()+timer,protocolo_timer);
}
```

Código Protocolo 3.4 Implementación de la función *star_timer()* de la clase *Protocolo3Sender.*

- **stop_timer():** Función que utiliza a la función *cancelEvent()* anteriormente explicada donde cancela la llegada del objeto *protocolo_timer*. La función *stop_timer()* se presenta en el Código Protocolo 3.5.

```
void Protocolo3Sender::stop_timer()
{    ev<<name()<<" Detiene el temporizador\n";
     // Se cancela el temporizador
     cancelEvent(protocolo_timer);
}
```

Código Protocolo 3.5 Implementación de la función *stop_timer()* de la clase *Protocolo3Sender***.**

- **initialize():** Función que empieza con la transmisión; en ella se crea únicamente la primera trama a la cual se le asigna un número de secuencia, posteriormente se crea una copia de la trama para enviar a la capa física ya que la original se la debe mantener en caso que sea necesario una retransmisión, finalmente se programa un temporizador en cuyo intervalo de tiempo el módulo emisor deberá recibir una confirmación de recepción de la trama enviada. En Código Protocolo 3.6 se presenta su implementación de la función *initialize()*.

```
void Protocolo3Sender::initialize()
{
     // Reserva un espacio de memoria para el puntero s
           s=new frame("Datos");
     // Se toma los valores configurados en el archivo de
     // inicialización .ini o en el archivo .ned
           frameSize= par ("frameSize");
           timer= par ("timer");
           loss = par("loss");
```

```
      // Se inicializa la variable
            next_frame_to_send=0;
      // Función que imprimirá en pantalla
      // el nombre del nodo y el tiempo
            registro_evento();
      // Se obtiene un paquete de la capa de red
            from_network_layer(&buffer);
      // Se coloca el paquete en el campo info de la trama
            s->setInfo(buffer);
      // Se inserta el número de secuencia en la trama
            s->setSeq(next_frame_to_send);
      // Se asigna el tamaño de la trama
            s->setLength(frameSize);
      // Se crea una copia de s para enviar
            frame *copia=(frame *)s->dup();
      // Se envía una copia de s a la capa física
            to_physical_layer(copia);
      // Se inicia el temporizador
            start_timer(protocolo_timer);
}
```

Código Protocolo 3.6 Implementación de la función *initialize()* de la clase *Protocolo3Sender*.

- **inc():** Función que incrementa en uno el valor de la variable obtenida del argumento hasta que llegue a cierto límite (MAX_SEQ); al llegar al límite, el valor de la variable retorna a cero y nuevamente se comienza a incrementar su valor. En el Código Protocolo 3.7 se muestra su implementación.

```
void Protocolo3SenderAgent::inc(int& k)
{
      if(k<MAX_SEQ)
      {
            k=k+1;
      }
      else
      {
            k=0;
      }
}
```

Código Protocolo 3.7 Implementación de la función *inc()* de la clase *Protocolo3Sender*.

- **retransmision():** Función que será invocada cuando se ha recibido al objeto que representa al temporizador. Aquí se crea un duplicado de la trama *s* anteriormente enviada, para lo cual se utiliza la función *dup()* de la clase *cSimpleModule.*

  La función *dup()* reserva espacio de memoria, crea y retorna una copia exacta del objeto, en este caso se crea un duplicado de la trama *s*.

En el Código Protocolo 3.8 se muestra la implementación de la función *retransmision()*.

```
void Protocolo3Sender::retransmision()
{
     // Se copia los valores de los campos de la trama
     // que anteriormente fue enviada
           frame *copia=(frame *)s->dup();

     // Se envía la trama a la capa física
           to_physical_layer(copia);

     // Se inicia el temporizador
           start_timer(protocolo_timer);
}
```

Código Protocolo 3.8 Implementación de la función *retransmisión()* de la clase *Protocolo3Sender*.

- **handleMessage():** Función que recibe como argumento un puntero a un temporizador expirado o a una trama de confirmación. En caso de que se reciba al temporizador expirado se invoca a la función *retransmision()* para reenviar la trama perdida. Para el segundo caso se verifica si el número de confirmación de la trama recibida (*r->getAck()*) es igual que el número de secuencia de la última trama que envió (*next_frame_to_send)*, si ésto ocurre se detiene el temporizador (*stop_timer()*), se obtiene un nuevo paquete de la capa de red y se incrementa el número de secuencia con la función *inc()*; posteriormente, se crea una nueva trama para enviar a la capa física, finalmente se borra la trama recibida. En el Código Protocolo 3.9 se presenta su implementación.

```
void Protocolo3Sender::handleMessage(cMessage *p_r_)
{

// Se verifica si el evento recibido es enviado por sí mismo,
// es decir si corresponde a la expiración de un temporizador
     if(p_r_->isSelfMessage())
     {
           // Se imprime en pantalla
           // el nombre del nodo y el tiempo
           registro_evento();
           ev<<name()<<" !!! Temporizador expirado "<<endl;
           // Mensaje que se visualizará en Tkenv
           // en el caso que ocurra la expiración del temporizador
           bubble("Temporizador expirado");
           //    Se inicia la retransmisión de la trama perdida
           retransmision();
     }
```

```
    else
    {
          r = check_and_cast<frame*>(p_r_);
          // Función que imprimirá en pantalla
          // el nombre del nodo y el tiempo
          registro_evento();
          ev<<name()<<" Recibe el ack de la trama:"<<
          r->getAck()<<std::endl;
          /*
          Si la trama que llega tiene el número de ack
          esperado, se detiene el temporizador y se obtiene un nuevo
          paquete de la capa de red
          */
          if(r->getAck()==next_frame_to_send)
          {
                    // Se detiene al temporizador
                    stop_timer();
                    // Se obtiene un paquete de la capa de red
                    from_network_layer(&buffer);
                    // Se cambia el valor de la siguiente trama que
                    // se va a enviar
                    inc(next_frame_to_send);
          }

          // Borra la trama recibida
          delete p_r_;

          // Se coloca el paquete en el campo info de la trama
          s->setInfo(buffer);

          // Se inserta el número de secuencia en la trama
          s->setSeq(next_frame_to_send);

          // Se asigna el tamaño de la trama
          s->setLength(frameSize);

          // Se crea una copia de s para enviar
          frame *copia=(frame *)s->dup();

          // Se envía una copia de s a la capa física
          to_physical_layer(copia);
          // Se inicia el temporizador
          start_timer(protocolo_timer);
    }
}
```

Código Protocolo 3.9 Implementación de la función *handleMessage()* de la clase *Protocolo3Sender.*

Clase Protocolo3Receiver

El objetivo de esta clase consiste en recibir una trama de datos y generar una trama de control que confirme la recepción.

En el Código Protocolo 3.10 se presenta la definición de la clase *Protocolo3Receiver.*

```
class Protocolo3Receiver:public cSimpleModule
{
private:
     frame *s;              // Variable que representa a la trama
                            // de confirmación que será enviada

     frame *r;              // Variable que representa a la trama
                            // de datos que será recibida


     int frameSize;         // Tamaño de la trama

     seq_nr frame_expected; // Variable que representa
                            // al número de secuencia de
                            // la trama que se espera recibir

     double channel_behavior;      // Variable que representa el
                                   // comportamiento del canal a través
                                   // de una distribución de
                                   // probabilidad, en base a esta se
                                   // conoce cuando se generan
                                   // pérdidas

     double loss;                  // Variable que representa la tasa de
                                   // pérdida en el canal considerando
                                   // el valor de channel_behavior

public:
// Constructor
     Protocolo3Receiver();

// Destructor
     virtual ~Protocolo3Receiver();

// Funciones del protocolo
     void to_network_layer(packet *msg);      // Envía un paquete a
                                              // la capa de red

     void to_physical_layer(frame * envio);   // Envía una trama
                                              // hacia la capa física

//Funciones auxiliares
     void registro_evento(); // Imprime el nombre del módulo
                             // y el tiempo en que ocurre un
                             // evento

     void inc(int& k);       // Incrementa circularmente
                             // el valor de la trama que se está
                             // esperando enviar o recibir

protected:

//Funciones que se redefinen de la clase cSimpleModule
     virtual void initialize();
     virtual void handleMessage(cMessage *p_r_);

};
```

Código Protocolo 3.10 Definición de la clase *Protocolo3Receiver.*

Variables

- **s:** Variable que representa a la trama de control generada para la confirmación de recepción, en ésta sólo se configurará el campo *ack_var* de la trama, con el cual se le indica al emisor que trama llegó correctamente.
- **r:** Variable que representa la trama que será recibida.
- **frame_expected:** Variable que representa el número de secuencia de la trama que se espera recibir.

Funciones

- **handleMessage():** Función que es invocada al llegar una trama al receptor en donde se verificará si el número de secuencia de la trama que recibe (*r->getSeq()*) corresponde a la trama que esperaba recibir (*frame_expected*); si esto se cumple se obtiene el paquete, se lo envía a la capa de red, se cambia el valor de *frame_expected*, posteriormente se envía la trama de confirmación a la que se asigna el número de acuse de recibo correspondiente. Su implementación se encuentra en el Código Protocolo 3.11.

```
void Protocolo3Receiver::handleMessage(cMessage *p_r_)
{
     r = check_and_cast<frame*>(p_r_);
     // Función que imprimirá en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();

     ev<<name()<<" Recibe la trama con seq:"<<r->getSeq()<<std::endl;

     // Se verifica si el número de secuencia de la trama recibida es la
     // misma que la esperada
           if(r->getSeq()==frame_expected)
           {
                 // Se envía el paquete a la capa de red
                 to_network_layer(&r->getInfo());
                 // Se cambia el valor de la trama esperada
                 inc(frame_expected);
           }

           // Borra la trama recibida
           delete p_r_;

           // Se crea una nueva trama para enviar como confirmación
           s=new frame("Confirmación");


           // Se inserta el número de confirmación de la trama
```

```
            s->setAck(1-frame_expected);
            // Se asigna el tamaño de la trama
            s->setLength(frameSize);

            // Se envía la trama a la capa física
            to_physical_layer(s);
}
```

Código Protocolo 3.11 Implementación de la función *handleMessage()* de la clase *Protocolo3Receiver*.

1.4.3.2.3. Canal con Ruido

Para la simulación de un canal con ruido es necesario implementar un modelo que simule el comportamiento del canal utilizando las distribuciones de probabilidad existentes en OMNET++ y, a través de la variable *loss* se indicará la tasa de pérdida de tramas en dicho canal.

El modelo es implementado en la función *to_physical_layer()* tanto para el módulo emisor así como para el receptor, en esta función se compara el valor generado para la variable *channel_behavior* (configurado en el archivo omnetpp.ini) con el valor asignado a la variable *loss,* si se cumple la condición se borra la trama que se está tratando de enviar caso contrario se la envía sin problemas. Cabe recalcar que cada vez que se invoque a la función *to_physical_layer()* el valor de la variable *channel_behavior* será diferente, además el valor dependerá de la distribución de probabilidad *uniform(0,1)* configurada en el archivo omnetpp.init. El modelo implementado se lo presenta en el Código Protocolo 3.12.

```
void Protocolo3Sender::to_physical_layer(frame *envio)
{
     // Se asigna a la variable channel_behavior el valor generado
     // en el archivo omnetpp.ini, el valor será diferente
     // para cada vez que se invoque la esta función
     channel_behavior = par ("channel_behavior");

     // Si se cumple la condición en que el valor generado para la
     // variable channel_behavior es menor al asignado a loss, entonces
     // se elimina la trama caso contrario se la envía sin problemas
          if(channel_behavior<loss)
          { ev<<" Se pierde la trama "<<endl;
               // Se muestra en Tkenv el texto
               // Trama Perdida
               bubble(" Trama Perdida");
               // Se borra la trama
               delete envio;
          }
          else
          {
```

```
                    // Envía una copia de la trama que recibe desde la
                    // capa de enlace

                     ev<<" Envía la trama con seq :"
                    <<envio->getSeq()<<endl;

                     send(envio,"salida");

               }

}
```

Código Protocolo 3.12 Implementación del modelo de pérdida de tramas en la función *to_physical_layer()* del módulo emisor.

Cabe indicar que el modelo de error para el módulo receptor es el mismo que el indicado en el Código Protocolo 3.12.

Distribuciones

Como antes se indicó, se ha modelado el comportamiento del canal de comunicaciones mediante la utilización de distribuciones de probabilidad que permiten generar valores aleatorios.

Las principales distribuciones utilizadas en OMNET++ son:

- *Uniform*
- *Exponential*
- *Normal*
- *Erlang*

Para mayor detalle se puede revisar la referencia **[1]**.

Las distribuciones pueden ser utilizadas en C++, en el lenguaje NED o en el archivo de configuración omnetpp.ini. Para esta implementación se ha configurado la distribución *uniform()* la cual ha sido asignada a la variable *channel_behavior* en el archivo omnetpp.ini.

1.4.3.3. Configuración del escenario de simulación

1.4.3.3.1. Configuración del escenario de simulación en el lenguaje NED

```
// Definición del módulo emisor
simple Protocolo3Sender

    parameters:
        frameSize: numeric,  // Tamaño de la trama
        timer: numeric,      // Tiempo en que expirará el temporizador
        loss: numeric,       // Tasa de pérdida de tramas
        channel_behavior: numeric; // Comportamiento del canal

    gates:
        in: entrada;         // Compuerta de entrada
        out: salida;         // Compuerta de salida

endsimple

// Definición del módulo receptor
simple Protocolo3Receiver

    parameters:
        frameSize: numeric,        // Tamaño de la trama
        loss: numeric,             // Tasa de pérdida de tramas
        channel_behavior: numeric; // Comportamiento del canal

    gates:
        in: entrada;               // Compuerta de entrada
        out: salida;               // Compuerta de salida

endsimple


module Protocolo3

    parameters:
        propagacion: numeric,      // Tiempo de propagación
        vtx: numeric,              // Velocidad de transmisión
        frameSize: numeric,
        timer: numeric,
        loss: numeric,
        channel_behavior: numeric;

    submodules:
        A: Protocolo3Sender;       // A: Instancia del módulo
                                   // Protocolo3Sender

            parameters:
                frameSize = frameSize,
                timer = timer,
                loss = loss,
                channel_behavior = channel_behavior;
            display: "p=35,96;i=device/pc,cyan";



        B: Protocolo3Receiver;     // B: Instancia del módulo
                                   // Protocolo3Receiver
           parameters:
```

```
                frameSize = frameSize,
                loss = loss,
                channel_behavior = channel_behavior;
            display: "p=224,96;i=device/pc,gold";

    connections:
        A.salida --> delay propagacion datarate vtx --> B.entrada;
        B.salida --> delay propagacion datarate vtx --> A.entrada;
    display: "b=250,250";

endmodule

network Network : Protocolo3
endnetwork
```

Escenario Protocolo3 Configuración del escenario para simular el protocolo "*simplex* para un canal con ruido".

1.4.3.3.2. Configuración de los parámetros en el archivo omnetpp.ini

En este archivo, se configurarán los valores para los parámetros que representan: el tiempo para la expiración del temporizador (*timer*), la tasa de pérdida de las tramas (*loss*) y, el comportamiento del canal (*channel_behavior*); el valor para el comportamiento del canal depende de la función de distribución de probabilidad *uniform(0,1).* En la Figura 2.35 se presenta el contenido del archivo omnetpp.ini para el protocolo "*simplex* para un canal con ruido".

```
[General]
network=Network

sim-time-limit = 1s

[Parameters]
Network.propagacion=0.050  # Tiempo de propagación en el canal en s
Network.vtx=10000000       # Velocidad de transmisión en bits por s
Network.frameSize=8000     # Tamaño de la trama en bits

Network.timer=0.120        # Tiempo para que el temporizador expire
Network.loss=0.01          # Tasa de pérdida de tramas

Network.channel_behavior=uniform(0,1) # Comportamiento del canal
                                      # de comunicación

[Run 1]
description="Protocolo3"
```

Figura 2.35 Configuración de los parámetros en omnetpp.ini para el protocolo "*simplex* para un canal con ruido".

1.4.3.4. Resultados de la simulación

1.4.3.4.1. Impresión en pantalla

A continuación, en la Figura 2.36 se indican los resultados que se imprimen en la pantalla al realizar la simulación del protocolo "*simplex* para un canal con ruido".

```
NODO:A, TIME:0 ms
 La capa de red envía un paquete
 Envía la trama con seq : 0
 Iniciando el temporizador
** Event #0.  T=0.0508000000 ( 50ms).  Module #3 `Network.B'
-----------------------------------------------
NODO:B, TIME:50.8 ms
 Recibe la trama con seq : 0
 La capa de red recibe : Nuevo Paquete
 Envía el ack de la trama : 0
** Event #1.  T=0.1016000000 (101ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:101.6 ms
 Recibe el ack de la trama : 0
 Detiene el temporizador
 La capa de red envía un paquete
 Envía la trama con seq : 1
 Iniciando el temporizador
** Event #2.  T=0.1524000000 (152ms).  Module #3 `Network.B'
-----------------------------------------------
NODO:B, TIME:152.4 ms
 Recibe la trama con seq : 1
 La capa de red recibe : Nuevo Paquete
 Envía el ack de la trama : 1
** Event #3.  T=0.2032000000 (203ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:203.2 ms
 Recibe el ack de la trama : 1
 Detiene el temporizador
 La capa de red envía un paquete
 Pierde la trama
 Iniciando el temporizador
** Event #4.  T=0.3232000000 (323ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:323.2 ms
 !!! Temporizador expirado
 Envía la trama con seq : 0
 Iniciando el temporizador
```

Figura 2.36 Impresión en pantalla obtenida de la simulación del protocolo "*simplex* para un canal con ruido".

En los resultados presentados en la Figura 2.36 se puede apreciar:

- El módulo emisor al tiempo 0ms envía una trama con secuencia 0 e inicia el temporizador; al tiempo 50.8ms el receptor recibe la trama enviada por el emisor, obtiene el paquete del campo de datos y lo pasa a la capa de red, posteriormente, envía una trama de confirmación de la trama recibida. Al tiempo 101.6ms el módulo emisor recibe la trama de

confirmación de la última trama enviada, por tanto detiene el temporizador y obtiene un nuevo paquete de la capa de red.

- Al tiempo 203.2ms se puede observar la pérdida de la trama que envió el módulo emisor y para la cual inicio un temporizador en espera de su confirmación. Al tiempo 323.2ms el temporizador iniciado concluye por lo que el nodo emisor procede a reenviar la trama para la cual esperaba su confirmación.

1.4.3.4.2. Visualización gráfica de la simulación

A continuación, en la Figura 2.37, Figura 2.38, Figura 2.39 y Figura 2.40 se presentan los resultados que se visualizan con la simulación del protocolo "*simplex* para un canal con ruido".

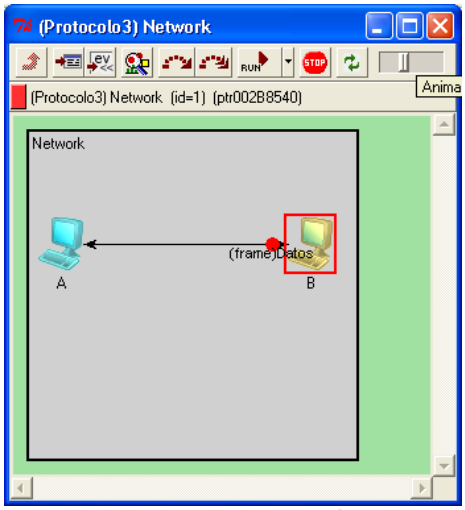


Figura 2.37 Envío de una trama de datos del módulo emisor al módulo receptor.

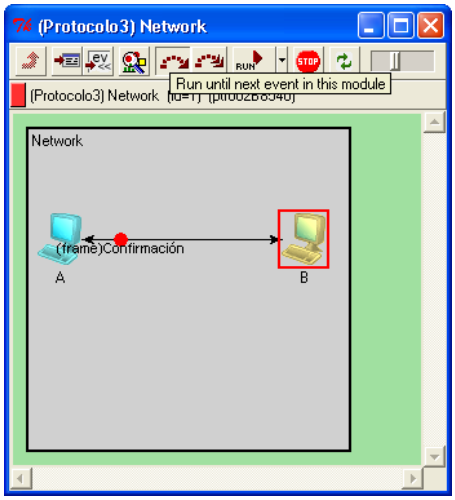


Figura 2.38 Envío de una trama de confirmación del módulo receptor al módulo emisor.

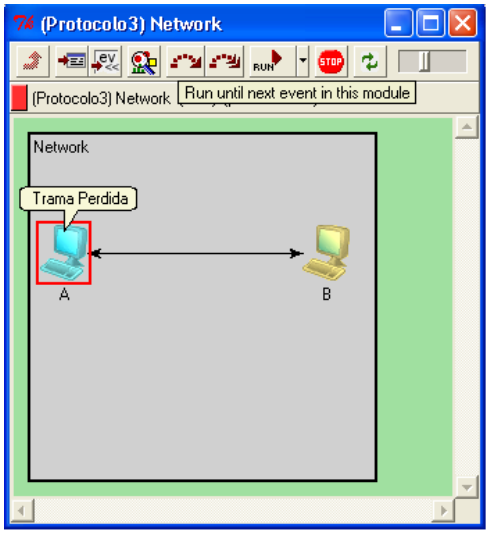


Figura 2.39 Pérdida de una trama de datos en el canal de comunicaciones.

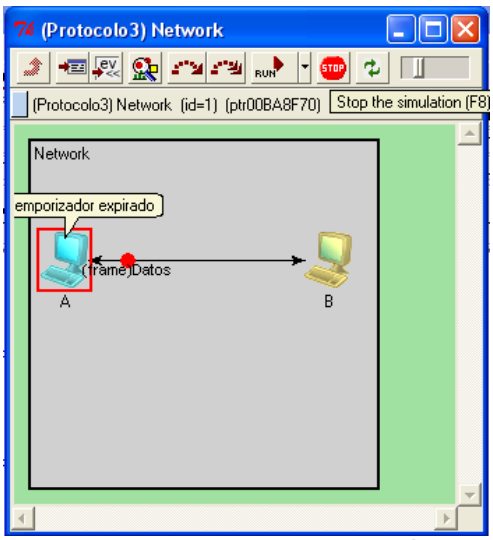


Figura 2.40 Expiración de un temporizador y retransmisión de la trama de datos perdida.

De los resultados obtenidos se puede concluir que el protocolo cumple con las consideraciones descritas para su implementación, es así que si la trama llega a su destino, esta es aceptada y se devuelve una trama de confirmación por ésta.En el caso que se pierda la trama, el temporizador para ésta expira y por tanto se realiza la retransmisión para dicha trama.

1.5. IMPLEMENTACIÓN DE PROTOCOLOS DE VENTANA CORREDIZA

Para la implementación de los protocolos de ventana corrediza se tienen las siguientes consideraciones:

- El emisor mantiene un grupo de números de secuencia que corresponde a las tramas que tiene permitido enviar, las mismas que caen dentro de la llamada ventana emisora. De igual manera, el receptor mantiene una ventana receptora, correspondiente al grupo de tramas que tiene permitido aceptar.
- El canal de comunicación es bidireccional.
- Se implementará un mecanismo denominado de superposición (*piggybacking*[1]).
- Los paquetes que se entregan a la capa de red deben estar en orden.

La principal característica para estos protocolos es la determinación del tamaño de la ventana.

Tamaño de la ventana

Para implementar estos protocolos necesariamente se debe considerar un tamaño de ventana; por lo tanto, cada trama de salida contiene un número de secuencia que va desde 0 hasta algún número máximo (2^n-1), por lo que el número de secuencia encaja perfectamente en un campo de *n* bits **[7]**.

1.5.1. PROTOCOLO DE VENTANA CORREDIZA DE UN BIT

1.5.1.1. Descripción del Protocolo

Las características que se consideran para este protocolo son:

- Transmisión de datos bidireccional.
- Tamaño de la ventana igual a 1.

[1] Mecanismo para enviar en conjunto una trama de datos con una de confirmación.

- Utiliza parada y espera, ya que el emisor envía una trama y espera su confirmación de recepción antes de continuar con la transmisión.
- Si no hay confirmación de recepción, se procede con la retransmisión.

Además de las características descritas, en este protocolo se plantean los siguientes escenarios:

1. Cuando una de las dos capas de enlace de datos comienza con la transmisión.
2. Si ambas capas inician la transmisión de forma simultánea.

En el primer caso puede surgir un escenario problema como el que se describe a continuación.

Suponga que la máquina A está tratando de enviar su trama 0 a la máquina B y que B está tratando de enviar su trama 0 a A. Suponga que A envía una trama a B, pero el intervalo de temporización de A es un poco corto. En consecuencia, A podría terminar su temporización repetidamente, enviando una serie de tramas idénticas.

Al llegar la primera trama válida a B, es aceptada y se procede a esperar la siguiente. Todas las tramas subsiguientes serán rechazadas, pues B espera una trama diferente a las tramas que llegan pues son duplicadas; por esa razón, B no extraerá un nuevo paquete de su capa de red.

En la Figura 2.41 se presenta el caso en el que en la máquina A se programa un temporizador con un intervalo de tiempo demasiado corto, por tanto su expiración es temprana provocando la retransmisión de la trama; en la figura se puede observar que la máquina B no acepta los paquetes duplicados.

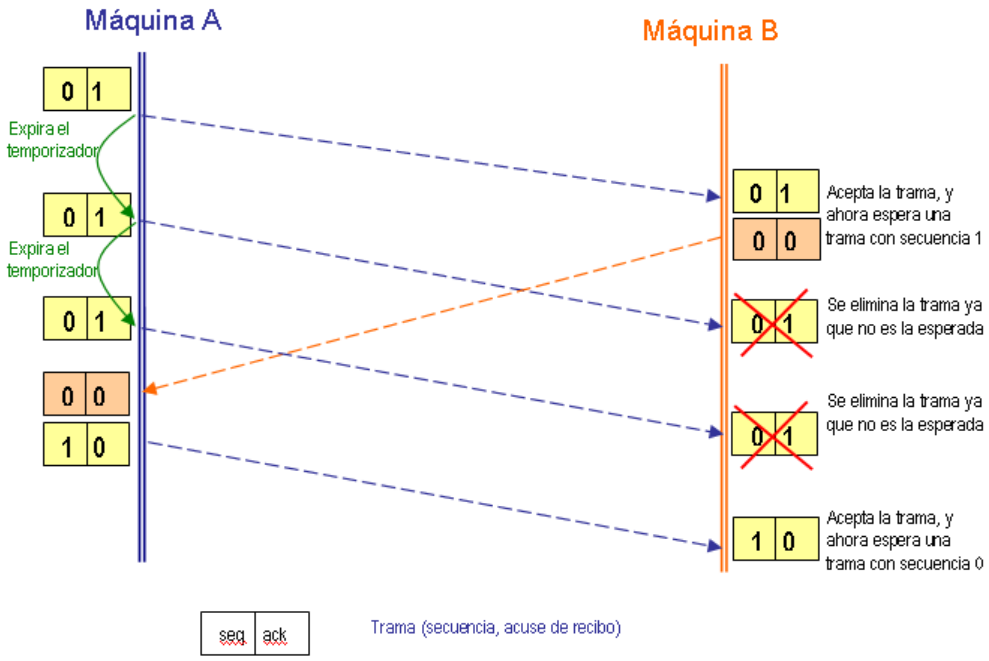


Figura 2.41 Caso de transmisión cuando el temporizador es demasiado corto.

Para el segundo caso puede surgir un problema de sincronización, en el que si A y B inician la comunicación simultáneamente, se cruzan sus primeras tramas y las capas de enlace de datos entran en una situación en la que la mitad de las tramas contienen duplicados, aún cuando no hay errores en la transmisión. Pueden ocurrir situaciones similares si ocurren varias expiraciones prematuras de temporizadores, aún cuando un lado comience primero (ver la Figura 2.42).

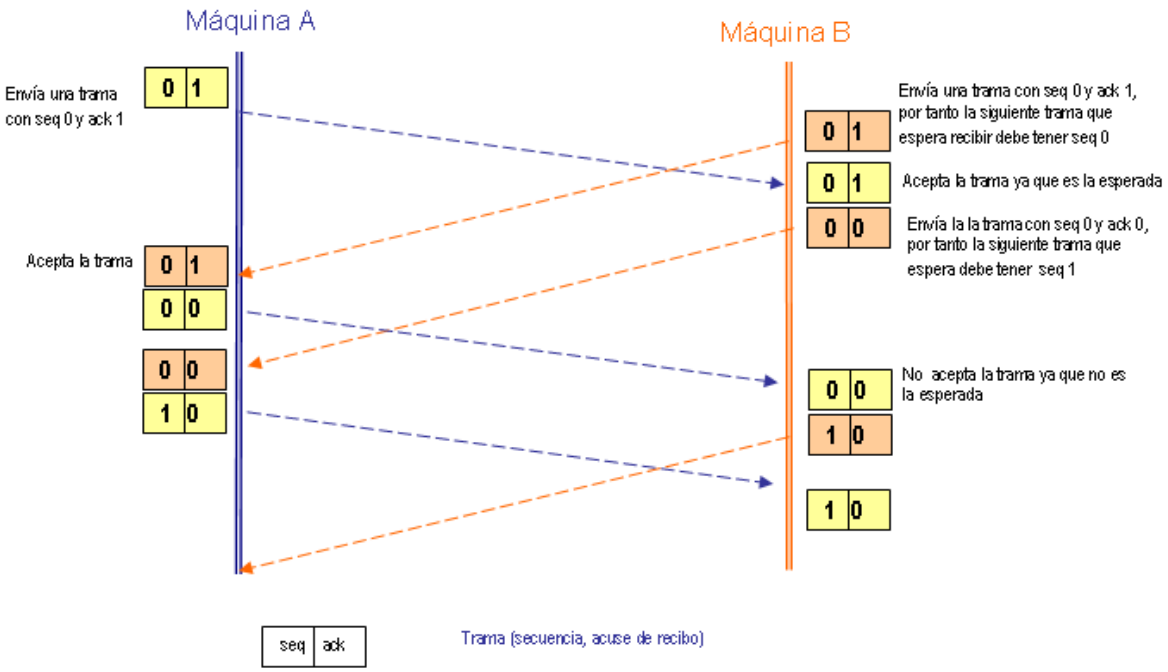


Figura 2.42 Caso de transmisión cuando las dos máquinas empiezan simultáneamente.

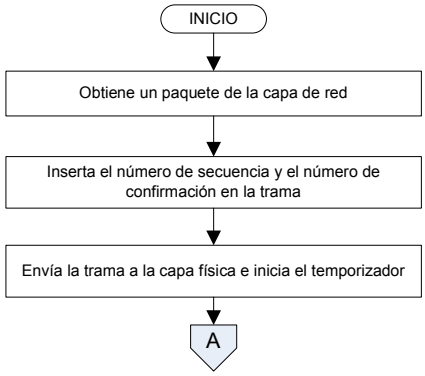
1.5.1.2. Diseño e Implementación en C++

Para la implementación se ha considerado los dos escenarios planteados en la descripción del protocolo de “ventana corrediza de un bit”.

La máquina que arranca obtiene el primer paquete de su capa de red, lo almacena en el búfer, lo inserta en el campo de datos de la trama que será enviada. Al llegar la trama al receptor, la capa de enlace de datos revisa el número de secuencia para saber si es un duplicado o no, igual que en el protocolo “*simplex* para un canal con ruido”. Si la trama es la esperada, se obtiene el paquete y se envía a la capa de red; además se cambia el valor de la trama a esperar.

El campo de confirmación de recepción de la trama recibida contiene el número de la última trama recibida sin error en la otra máquina. Si este número concuerda con el valor de la siguiente trama que está tratando de enviar el emisor (a su vez será el número de secuencia de esta trama); entonces éste sabrá que la última trama enviada ha llegado correctamente al receptor, por tanto, puede obtener el siguiente paquete de su capa de red para almacenarlo en el búfer. Si el valor del campo de confirmación no concuerda, deberá enviar una copia de la última trama enviada.

Debido a que la transmisión de datos es bidireccional, se ha implementado un solo módulo que cumpla con el rol de emisor y receptor. En la Figura 2.43 se presenta el diagrama de flujo para el protocolo de “ventana corrediza de un bit”.



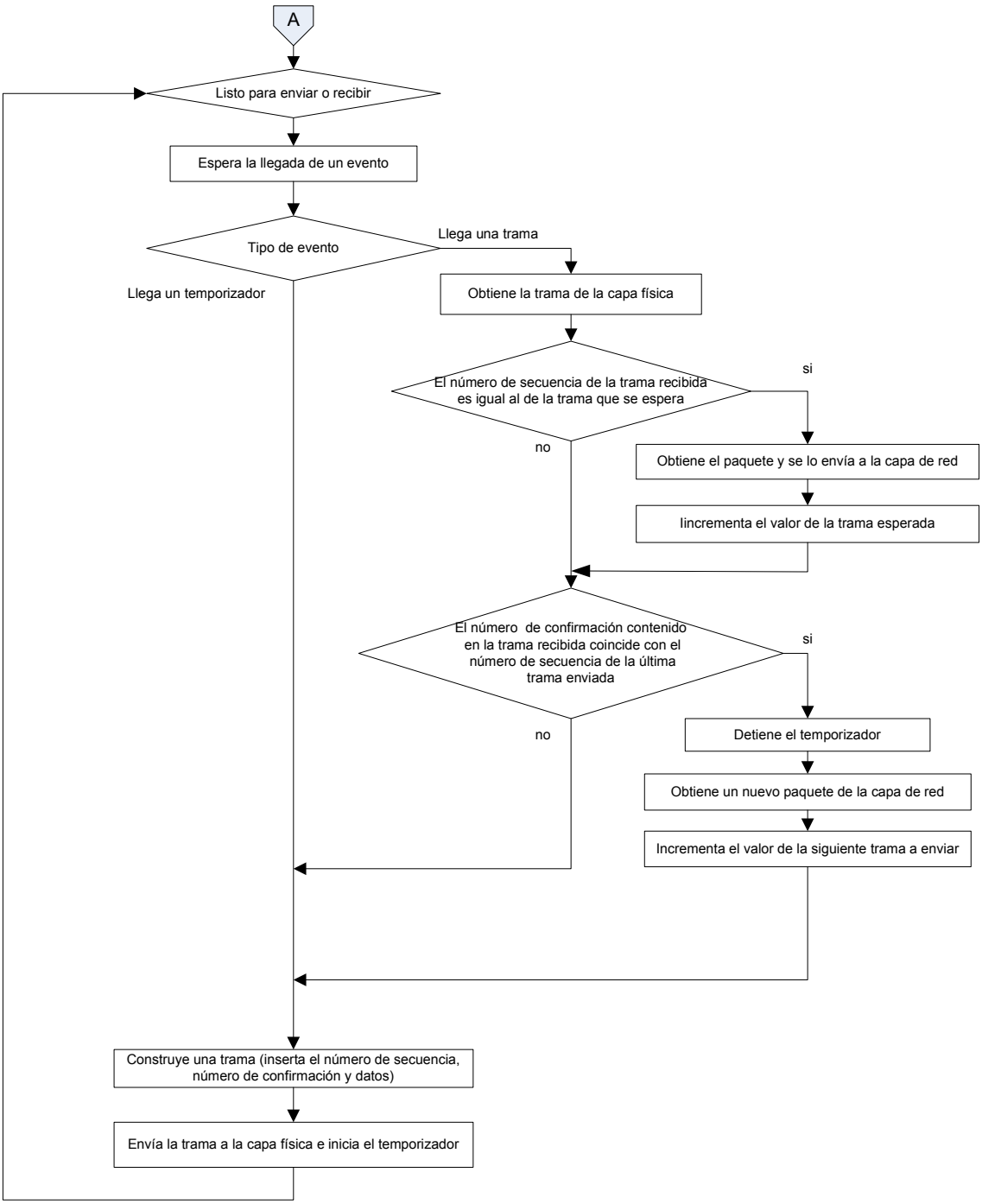


Figura 2.43 Diagrama de flujo para el protocolo de “ventana corrediza de un bit”.

Al igual que en el Protocolo 3, para solucionar el problema que surge cuando existe pérdida de tramas, se han implementado temporizadores con la misma lógica y funcionalidad.

1.5.1.2.1. Módulo

Clase Protocolo4

El objetivo de esta clase es representar a un módulo que cumpla con el rol de emisor y receptor a la vez. En el Código Protocolo 4.1 se presenta la definición de la clase *Protocolo4.*

```
// Máximo número que tendrá la secuencia
#define MAX_SEQ 1

// Enum que es utilizado para distinguir a los temporizadores de
// las tramas

typedef enum{FRAME=1,TIMEOUT=2} tipo_evento;
typedef int seq_nr;

class Protocolo4: public cSimpleModule
{
private:

// Variables del protocolo
     frame *s;               // Variable que representa a la trama
                             // que será enviada
     frame *r;               // Variable que representa a la trama
                             // que será recibida
     packet buffer;          // Paquete que se obtiene  de la
                             // capa de red
     int frameSize;          // Tamaño de la trama

     double timer;           // Variable que representa el
                             // intervalo de tiempo que espera
                             // antes que el temporizador expire
     seq_nr next_frame_to_send;    // Variable que representa
                                   // el número de secuencia de
                                   // la siguiente trama de salida
     seq_nr frame_expected;        // Variable que representa
                                   // al número de secuencia de
                                   // la trama que espera recibir
     cMessage *protocolo_timer;    // Variable que representa
                                   // al temporizador
     const char* case_;            // Variable que permite escoger el
                                   // módulo que iniciará con la
                                   // transmisión, o si iniciarán los
                                   // dos módulos simultáneamente
     double channel_behavior;      // Variable que representa el
                                   // comportamiento del canal a través
                                   // de una distribución de
                                   // probabilidad
     double loss;                  // Variable que representa la tasa
                                   // de pérdida de tramas en el canal

public:
// Constructor
     Protocolo4();
```

```
// Destructor
     virtual ~Protocolo4();

// Funciones del protocolo
     void from_network_layer(packet*  msg);    // Obtiene un paquete
                                               // de la capa de red
     void to_network_layer(packet *msg);       // Envía un paquete a
                                               // la capa de red
     void to_physical_layer(frame * envio);    // Envía una trama
                                               // hacia la capa física
// Funciones para el temporizador
     void start_timer(cMessage *timer);   // Inicia el temporizador
     void stop_timer();                    // Desactiva el temporizador

// Funciones auxiliares
     void crear_paquete(packet *msg);    // Genera un paquete en
                                         // la capa de red
     void registro_evento();             // Imprime el nombre del módulo
                                         // y el tiempo en que ocurre un
                                         // evento
     void inc(int& k);                   // Incrementa circularmente
                                         // el valor del argumento
     void retransmision();               // Retransmite una trama cuando
                                         // expira el temporizador
protected:
// Funciones de la clase cSimpleModule que se redefinen
    virtual void initialize();           // Función que es invocada para
                                         // iniciar la simulación
    virtual void handleMessage(cMessage *p_r_);// Función que recibe
                                               // la trama desde la
                                               // capa física
};
```

Código Protocolo 4.1 Definición de la clase *Protocolo4.*

Variables

- **case_:** Variable que permite al usuario escoger que módulo empezará con la transmisión de datos o a su vez si empezarán los dos módulos; de esta manera, se podrán simular los dos escenarios expuestos en la descripción del protocolo.
- **tipo_evento:** Enumeración definida con el objetivo de diferenciar una trama de un temporizador.

Funciones

- **initialize():** Función en la que se inicializan las variables y parámetros del módulo. Además en ésta se determina el escenario de simulación de acuerdo al valor configurado para la variable *case_.* Si el valor

configurado para la variable *case_* coincide con el nombre del módulo (*strcmp(case_, name()) == 0*) o con el *string* “dos”(*strcmp(case_, "dos") == 0*), entonces el módulo empezará a transmitir la primera trama, para lo cual obtendrá un paquete de la capa de red, y construirá una trama asignándole el número de secuencia, el número de confirmación; posteriormente, la enviará a la capa física e iniciará un temporizador para la espera de confirmación de la trama que se está enviando (ver el Código Protocolo 4.2).

```
void Protocolo4::initialize()
{
     // Reserva un espacio de memoria para el puntero s
           s=new frame("frame",FRAME);
     // Se toma los valores configurados en el archivo de
     // inicialización .ini o en el archivo .ned
           frameSize= par("frameSize");
           timer= par ("timer");
           case_ =  (const char *)par("case_");
           channel_behavior = par ("channel_behavior");
           loss = par("loss");
     // Inicializacion de variables del protocolo
           next_frame_to_send=0;
           frame_expected=0;
    // Se obtiene un paquete de la capa de red
           from_network_layer(&buffer);
     // Si el valor de la variable case_ coincide con el nombre del
     // módulo o con el string "dos", entonces el módulo empieza
     // con la transmisión
     if((strcmp(case_, name()) == 0) || (strcmp(case_, "dos") == 0))
     {
                 // Se imprime en pantalla
                 // el nombre del nodo y el tiempo
                 registro_evento();
                 ev<<" Empieza la transmisión "<<endl;
                 // Se inserta el número de secuencia en la trama
                 s->setSeq(next_frame_to_send);
                 // Se inserta el número de confirmación de la trama
                 s->setAck(1-frame_expected);
                 // Se coloca el paquete en el campo info de la trama
                 s->setInfo(buffer);
                 // Se asigna el tamaño de la trama
                 s->setLength(frameSize);
                 // Se crea una copia de s para enviar
                 frame *copia=(frame *)s->dup();
                 // Se envía una copia de s a la capa física
                 to_physical_layer(copia);
                 // Se inicia el temporizador
                 start_timer(protocolo_timer);
     }
}
```

Código Protocolo 4.2 Implementación de la función *initialize()* de la clase *Protocolo4*.

- **handleMessage():** Función que será invocada automáticamente por el *kernel* del simulador cuando llegue una trama o expire un temporizador.

  Si expira un temporizador se procede a la retransmisión de la última trama enviada mediante la invocación de la función *retransmision()*.

  Si llega una trama se verifica tanto el número de secuencia como el de confirmación de recepción para tomar una decisión, de esta manera se implementa la técnica de superposición *(piggybacking*).

  Al verificar si el número de secuencia de la trama recibida (*r–>getSeq()*) es igual que el número de secuencia que espera recibir (*frame_expected*) se procede a enviar el paquete a la capa de red y se cambia el valor de la variable *frame_expected* (de acuerdo al tamaño de la ventana), es decir si el valor de *frame_expected* era 0, se cambiará a 1, y si el valor era 1 se cambiará a 0.

  Posteriormente, se verifica si el número de confirmación (*r->getAck())* contenido en la trama recibida es el mismo que el número de secuencia de la última trama que fue enviada, cuyo valor asignado fue *next_frame_to_send*; si ese es el caso, se obtiene un nuevo paquete de la capa de red y se incrementa el valor de la variable *next_frame_to_send* (de la misma forma que se indicó para *frame_expected*). Además, se borra la trama recibida, se crea una nueva trama y se la envía a la capa física (ver el Código Protocolo 4.3).

```
void Protocolo4::handleMessage(cMessage *p_r_)
{
     registro_evento();
     // Se verifica si lo que ha llegado es un temporizador expirado
     // o una trama
     // Si se recibe un temporizador
     if(p_r_->kind()==TIMEOUT )
     {

           // Se muestra en Tkenv el texto
           // Temporizador expirado
           bubble("Temporizador expirado");
           ev<<"!!! Temporizador expirado"<<endl;
           // Se inicia la retransmisión de la trama perdida
           retransmision();

     }
```

```
    // Si se recibe una trama
    if(p_r_->kind()==FRAME)
    {
          // Convierte al objeto recibido
          // de tipo cMessage a tipo frame
          frame *r = check_and_cast<frame*>(p_r_);

          ev<<" Recibe la trama (seq,ack):("<<r->getSeq()<<","
           <<r->getAck()<<")\n";

          // Se verifica si cumple alguna de las dos condiciones
          if(r->getSeq()==frame_expected ||
           r->getAck()==next_frame_to_send)
          {
                /*Verifica si el número de secuencia de la trama
                recibida coincide con el número de la trama que
                esperaba recibir; si esto se cumple envía el paquete
                a la capa de red e incrementa el número de la
                trama esperada*/
                if(r->getSeq()==frame_expected)
                {

                      // Se envía el paquete a la capa de red
                      to_network_layer(&r->getInfo());

                      // Se cambia el valor de la trama esperada
                      inc(frame_expected);

                }
                 /*
                Verifica si el número del ack de la trama recibida
                coincide con el número de secuencia de última trama
                que fue enviada, si esto se cumple se detiene el
                temporizador para la trama, se obtiene un nuevo
                paquete de la capa de red y se cambia el valor de
                la siguiente trama que se va a enviar
                */
                if(r->getAck()==next_frame_to_send)
                {
                      // Se detiene al temporizador
                      stop_timer();

                      // Se obtiene un paquete de la capa de red
                      from_network_layer(&buffer);

                      // Se cambia el valor de la siguiente trama que
                      // se va a enviar
                      inc(next_frame_to_send);
                }
          }
          // Se borra la trama recibida
          delete p_r_;

          // Se inserta el número de secuencia en la trama
          s->setSeq(next_frame_to_send);

          // Se inserta el número de confirmación de la trama
          s->setAck(1-frame_expected);
```

```
            // Se coloca el paquete en el campo info de la trama
            s->setInfo(buffer);

            // Se asigna el tamaño de la trama
            s->setLength(frameSize);

            // Se inserta el número de confirmación de la trama
            frame *copia=(frame*)s->dup();

            // Se envía una copia de s a la capa física
            to_physical_layer(copia);

            // Se inicia el temporizador
            start_timer(protocolo_timer);
      }
}
```

Código Protocolo 4.3 Implementación de la función *handleMessage()* de la clase *Protocolo4.*

- **start_timer():** Función en la que se inicializa el temporizador, a diferencia de la función implementada en el protocolo "*simplex* para un canal con ruido", en este se debe verificar si el temporizador ha sido programado anteriormente (esto sucede cuando se recibe una trama que no contiene la confirmación de la última trama enviada), si éste es el caso se lo cancela para reprogramarlo. En el Código Protocolo 4.4 se presenta la implementación.

```
void Protocolo4::start_timer(cMessage *protocolo_timer_)
{
     // Se debe reprogramar el temporizador
     if(protocolo_timer_->arrivalTime()>simTime())
     {
           //Se cancela el temporizador anterior
           cancelEvent(protocolo_timer_);
     }

     ev<<" Iniciando el temporizador\n";
     scheduleAt(simTime()+timer,protocolo_timer_);

}
```

Código Protocolo 4.4 Implementación de la función *start_timer()* de la clase *Protocolo4*.

La función *stop_timer()* solo será invocada cuando llegue una trama que contenga la confirmación esperada.

1.5.1.3. Configuración del escenario de simulación

1.5.1.3.1. Configuración del escenario de simulación en el lenguaje NED

```
// Definición del módulo emisor y receptor
simple Protocolo4
     parameters:
           frameSize:numeric, // Tamaño de la trama
           timer:numeric,     // Tiempo en que expirará el temporizador
           channel_behavior:numeric,// Comportamiento del canal
           loss:numeric,// Tasa de pérdida de tramas
           case_:string;// Permite escoger entre los escenarios de
                        // simulación expuestos en la descripción
                        // del protocolo
    gates:
        out: salida;   // Compuerta de salida
        in: entrada;   // Compuerta de entrada
endsimple

module Protocolo4_

     parameters:
     propagacion:numeric,// Tiempo de propagación
     vtx:numeric,        // Velocidad de transmisión
     frameSize:numeric,
     timer:numeric,
     channel_behavior:numeric,
     loss:numeric,
     case_:string;
    submodules:
       A: Protocolo4; // A: Instancia del módulo
                      // Protocolo4
                parameters:
                frameSize=frameSize,
                timer=timer,
                case_=case_,
                loss=loss,
                channel_behavior=channel_behavior;
            display: "p=35,96;i=device/pc,cyan";
        B: Protocolo4; // B: Instancia del módulo
                      // Protocolo4
                parameters:
                frameSize=frameSize,
                timer=timer,
                case_=case_,
                loss=loss,
                channel_behavior=channel_behavior;
                 display: "p=224,96;i=device/pc,gold";
    connections:
        A.salida --> delay propagacion datarate vtx --> B.entrada;
        B.salida --> delay propagacion datarate vtx --> A.entrada;
        display: "b=250,250";
endmodule
network Network : Protocolo4_
endnetwork
```

Escenario Protocolo4 Configuración del escenario para simular el protocolo de “ventana corrediza de un bit”.

En éste protocolo se ha configurado el mismo tamaño de la trama en los dos módulos que participan en la simulación.

1.5.1.3.2. Configuración de los parámetros en el archivo omnetpp.ini

En este archivo se configuran los mismos parámetros que en los protocolos anteriores y, la variable *case_* con la cual el usuario podrá escoger el escenario de simulación que desee.

```
[General]
preload-ned-files=Protocolo4.ned
sim-time-limit = 1s
network=Network

[Parameters]
Network.propagacion=0.050    # Tiempo de propagación en el canal en s
Network.vtx=10000000         # Velocidad de transmisión en bits por s
Network.frameSize=8000       # Tamaño de la trama en bits
Network.timer=0.120          # Tiempo para que el temporizador
                             # expire
Network.loss=0.01            # Tasa de pérdida de tramas
Network.channel_behavior=uniform(0,1)   # Comportamiento del canal
                                        # de comunicación

[Run 1]
description="Protocolo4, empieza A"
Network.case_="A"
[Run 2]
description="Protocolo4, empieza B"
Network.case_="B"
[Run 3]
description="Protocolo4, empiezan los dos"
Network.case_=dos
```

Figura 2.44 Configuración de los parámetros en omnetpp.ini para el protocolo de "ventana corrediza de un bit".

1.5.1.4. Resultados de la simulación

1.5.1.4.1. Impresión en pantalla

Módulo A empieza con la transmisión

A continuación, en la Figura 2.45 se indican los resultados que se imprimen en la pantalla al realizar la simulación del protocolo "ventana corrediza de un bit", cuando el módulo A empieza con la transmisión.

```
La capa de red envía un paquete
--------------------------------------------------
NODO:A, TIME:0 ms
 Empieza la transmisión
 Envía la trama (seq,ack):(0,1)
 Iniciando el temporizador
 La capa de red envía un paquete
** Event #0.  T=0.0508000000 ( 50ms).  Module #3 `Network.B'
--------------------------------------------------
NODO:B, TIME:50.8 ms
 Recibe la trama (seq,ack): (0,1)
 La capa de red recibe: Nuevo Paquete
 Envía la trama  (seq,ack):(0,0)
 Iniciando el temporizador
** Event #1.  T=0.1016000000 (101ms).  Module #2 `Network.A'
--------------------------------------------------
NODO:A, TIME:101.6 ms
 Recibe la trama (seq,ack):(0,0)
 La capa de red recibe: Nuevo Paquete
 Detiene el temporizador
 La capa de red envía un paquete
 Pierde la trama
 Iniciando el temporizador
** Event #2.  T=0.1708000000 (170ms).  Module #3 `Network.B'
--------------------------------------------------
NODO:B, TIME:170.8 ms
!!! Temporizador expirado
 Envía la trama (seq,ack):(0,0)
 Iniciando el temporizador
** Event #3.  T=0.2216000000 (221ms).  Module #2 `Network.A'
--------------------------------------------------
NODO:A, TIME:221.6 ms
!!! Temporizador expirado
 Envía la trama (seq,ack): (1,0)
 Iniciando el temporizador
** Event #4.  T=0.2216000000 (221ms).  Module #2 `Network.A'
--------------------------------------------------
NODO:A, TIME:221.6 ms
 Recibe la trama (seq,ack):(0,0)
 Envía la trama (seq,ack):(1,0)
 Iniciando el temporizador
** Event #5.  T=0.2724000000 (272ms).  Module #3 `Network.B'
--------------------------------------------------
NODO:B, TIME:272.4 ms
 Recibe la trama (seq,ack):(1,0)
 La capa de red recibe: Nuevo Paquete
 Detiene el temporizador
 La capa de red envía un paquete
 Envía la trama (seq,ack):(1,1)
 Iniciando el temporizador
** Event #6.  T=0.2732000000 (273ms).  Module #3 `Network.B'
--------------------------------------------------
NODO:B, TIME:273.2 ms
 Recibe la trama (seq,ack): (1,0)
 Envía la trama (seq,ack): (1,1)
 Iniciando el temporizador
```

Figura 2.45 Impresión en pantalla obtenida de la simulación del protocolo “ventana corrediza de un bit”cuando el módulo A empieza la transmisión.

En la Figura 2.45 se puede observar que en una misma trama se transmite datos y control representados en los campos seq y ack de la trama (por ejemplo al tiempo 0s se transmite la trama con secuencia 0 y acuse de recibo para la trama 1), de igual manera que en el protocolo 3 se inicializa un temporizador por cada trama enviada.

Módulo A y B empiezan simulatáneamente con la transmisión

A continuación, en la Figura 2.46 se indican los resultados que se imprimen en la pantalla al realizar la simulación del protocolo "ventana corrediza de un bit" cuando los dos módulos empiezan con la transmisión.

```
 La capa de red envía un paquete
-------------------------------------------------
NODO:A, TIME:0 ms
 Empieza la transmisión
 Envía la trama (seq,ack):(0,1)
 Iniciando el temporizador
 La capa de red envía un paquete
-------------------------------------------------
NODO:B, TIME:0 ms
 Empieza la transmisión
 Envía la trama (seq,ack):(0,1)
 Iniciando el temporizador
** Event #0.  T=0.0508000000 ( 50ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:50.8 ms
 Recibe la trama (seq,ack):(0,1)
 La capa de red recibe: Nuevo Paquete
 Envía la trama (seq,ack):(0,0)
 Iniciando el temporizador
** Event #1.  T=0.0508000000 ( 50ms).  Module #2 `Network.A'
-------------------------------------------------
NODO:A, TIME:50.8 ms
 Recibe la trama (seq,ack):(0,1)
 La capa de red recibe: Nuevo Paquete
 Envía la trama (seq,ack):(0,0)
 Iniciando el temporizador
** Event #2.  T=0.1016000000 (101ms).  Module #2 `Network.A'
-------------------------------------------------
NODO:A, TIME:101.6 ms
 Recibe la trama (seq,ack):(0,0)
 Detiene el temporizador
 La capa de red envía un paquete
 Envía la trama (seq,ack):(1,0)
 Iniciando el temporizador
** Event #3.  T=0.1016000000 (101ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:101.6 ms
 Recibe la trama (seq,ack):(0,0)
 Detiene el temporizador
 La capa de red envía un paquete
 Envía la trama (seq,ack):(1,0)
 Iniciando el temporizador
```

El módulo A envía su primera trama

El módulo B envía su primera trama

El módulo B recibe la trama

El módulo A recibe la trama

```
** Event #4.  T=0.1524000000 (152ms).  Module #3 `Network.B'
-----------------------------------------------
NODO:B, TIME:152.4 ms
 Recibe la trama (seq,ack):(1,0)
 La capa de red recibe: Nuevo Paquete
 Envía la trama (seq,ack):(1,1)
 Iniciando el temporizador
** Event #5.  T=0.1524000000 (152ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:152.4 ms
 Recibe la trama (seq,ack):(1,0)
 La capa de red recibe: Nuevo Paquete
 Envía la trama (seq,ack):(1,1)
 Iniciando el temporizador
** Event #6.  T=0.2032000000 (203ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:203.2 ms
 Recibe la trama (seq,ack):(1,1)
 Detiene el temporizador
 La capa de red envía un paquete
 Envía la trama (seq,ack):(0,1)
 Iniciando el temporizador
** Event #7.  T=0.2032000000 (203ms).  Module #3 `Network.B'
-----------------------------------------------
NODO:B, TIME:203.2 ms
 Recibe la trama (seq,ack):(1,1)
 Detiene el temporizador
 La capa de red envía un paquete
 Envía la trama (seq,ack):(0,1)
 Iniciando el temporizador
.
.
```

Figura 2.46 Impresión en pantalla obtenida de la simulación del protocolo “ventana corrediza de un bit”cuando los dos módulos empiezan la transmisión.

En la Figura 2.46 se puede observar que el módulo A y B empiezan con la transmisión al mismo tiempo es decir a los 0s lo que provoca que reciba tramas duplicadas las cuales serán descartadas por ejemplo al tiempo 50.8ms el módulo B recibe una trama con número de secuencia 0 que es la que esperaba, por tanto obtiene el paquete y lo pasa a la capa de red, posteriormente al tiempo 101.6ms el módulo B vuelve a recibir una trama con secuencia 0 por lo que la descarta ya que es un duplicado.

1.5.1.4.2. Visualización gráfica de la simulación

Módulo A empieza con la transmisión

A continuación, desde la Figuras 2.47 hasta la Figura 2.50, se presentan los resultados que se visualizan con la simulación del protocolo “ventana corrediza de un bit” cuando el módulo A inicia la transmisión.

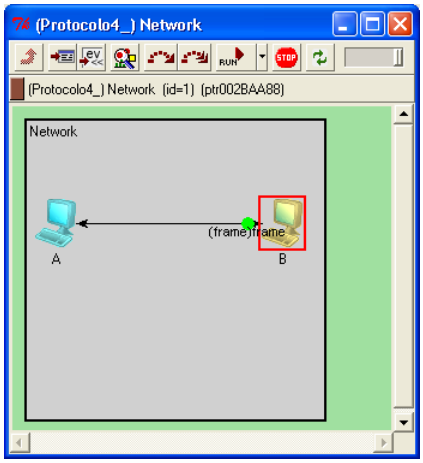


Figura 2.47 El módulo A envía una trama al módulo B.

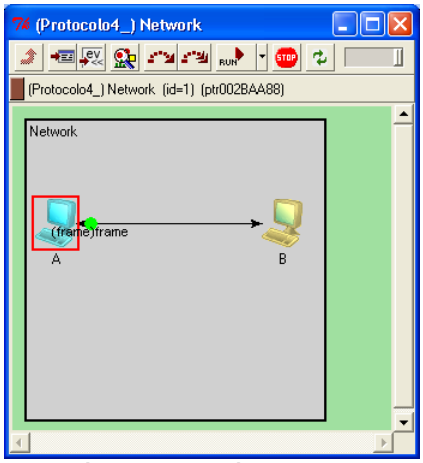


Figura 2.48 El módulo B envía una trama al módulo A.

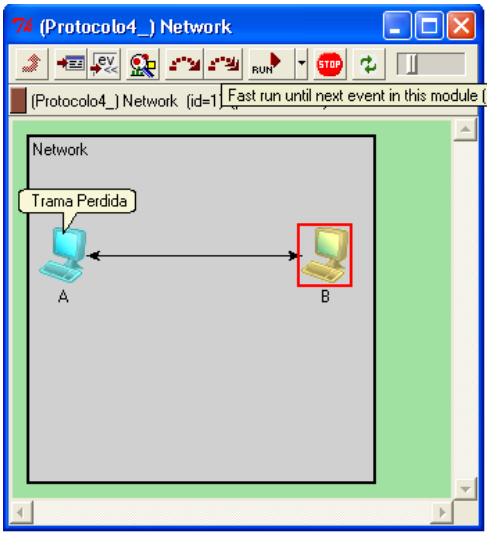


Figura 2.49 El módulo A pierde una trama.

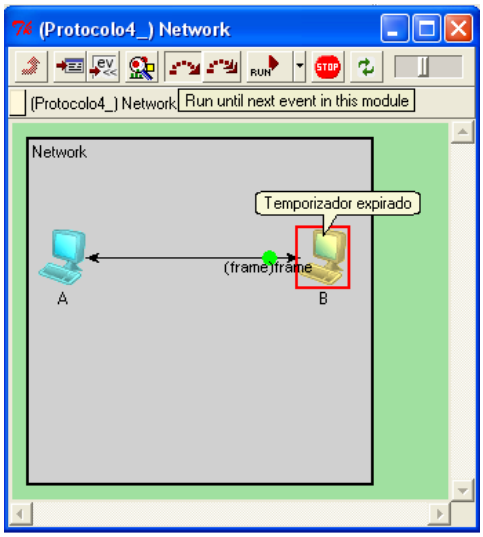


Figura 2.50 Expira el temporizador en el módulo B.

Módulos A y B empiezan simulataneamente con la transmisión

A continuación en la Figura 2.51 y la Figura 2.52 se presentan los resultados que se visualizan con la simulación del protocolo “ventana corrediza de un bit” cuando los módulo A y B inician la transmisión simultaneamente.

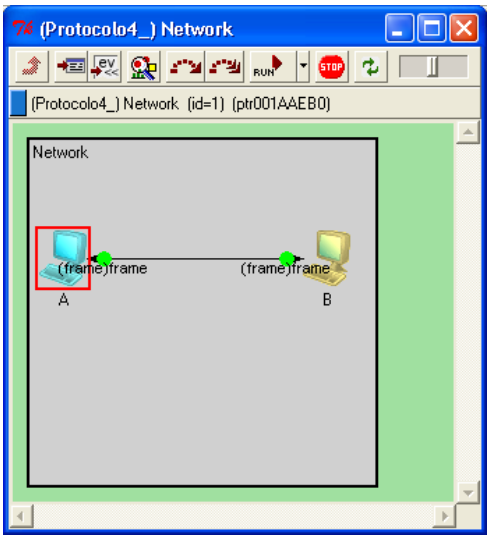


Figura 2.51 Los módulos A y B empiezan con la transmisión simultáneamente.

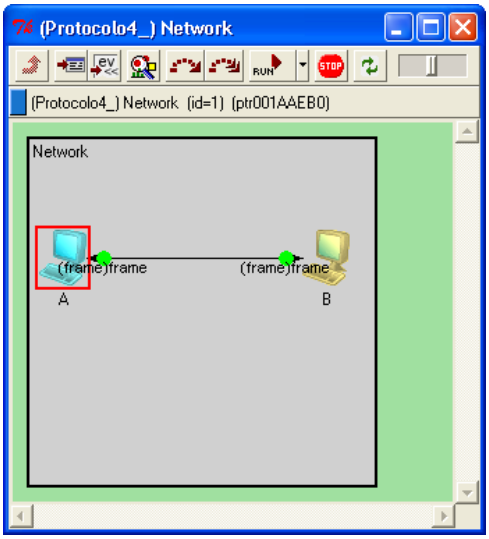


Figura 2.52 Los módulos A y B reciben tramas al mismo tiempo.

De los resultados presentados se puede concluir que el funcionamiento del protocolo es el esperado según los escenarios descritos en la Sección 2.2.1.1, así como también se puede observar la técnica de *piggybacking* implementada.

1.5.2. PROTOCOLO DE VENTANA CORREDIZA CON RETROCESO N

1.5.2.1. Descripción del protocolo

En este protocolo, a diferencia de los anteriores, se elimina el supuesto de que el tiempo requerido para que una trama llegue al receptor más el tiempo necesario para que la confirmación de recepción regrese es insignificante. En esta situación, el tiempo de viaje de ida y vuelta prolongado puede tener implicaciones importantes para la eficiencia del aprovechamiento del ancho de banda.

La solución para evitar el desperdicio de ancho de banda que surge al enviar una trama y esperar su confirmación antes de continuar con la siguiente transmisión, está en permitir que el emisor pueda enviar hasta *w* tramas antes de bloquearse, en lugar de enviar sólo una. Con una selección adecuada de *w*, el emisor podrá transmitir tramas continuamente durante el tiempo igual a *t* sin llenar la ventana.

$$t = \text{tiempo de propagación de ida} + \text{tiempo de propagación de vuelta} + \text{tiempo de transmisión de la trama de ida} + \text{tiempo de transmisión de la trama de vuelta}$$

Para explicar lo anteriomente descrito se considera un canal de comunicación con un ancho de banda de *b* bits/segundos, el tamaño de la trama de *l* bits y el tiempo

de propagación de ida y vuelta de una trama igual *R* segundos; por tanto el tiempo requerido para enviar una sola trama es de *l/b* segundos; una vez que ha sido enviado el último bit de una trama de datos, hay un retardo de *R/2* segundos antes de que llegue ese bit al receptor y un retardo de *R/2* segundos para que la confirmación de recepción llegue de regreso, lo que da un retardo toral de *R* segundos. Con las consideraciones anteriores para el protocolo de ventana corrediza de un bit, la línea estará ocupada durante *l/b* segundos e inactiva durante *R* segundos dando una utilización del canal:

$$Utilización\ del\ canal = \frac{\frac{l}{b}}{\frac{l}{b} + R} = \frac{l}{l + bR}$$

Por tanto, la necesidad de una ventana grande en el lado emisor se presenta cuando el producto del ancho de banda por el tiempo de propagación del viaje de ida y vuelta es grande. Si el ancho de banda es alto, incluso para un tiempo de propagación moderado, el emisor agotará su ventana rápidamente a menos que tenga una ventana grande. Si el tiempo de propagación es grande el emisor agotará su ventana incluso con un ancho de banda moderada. El producto de estos dos factores determina básicamente cuál es la utilización del canal, y el emisor debe estar en capacidad de llenarlo sin detenerse para poder funcionar con una eficiencia máxima **[7]**.

El envió de tramas consecutivamente por un canal de comunicación no confiable presenta el problema que se describe a continuación:

¿Qué ocurre si una trama, a la mitad de una serie larga de tramas transmitidas, se daña o se pierde? Llegarán grandes cantidades de tramas sucesivamente al receptor antes de que el emisor se entere de que algo anda mal. Cuando llega una trama dañada al receptor, obviamente debe descartarse, pero, ¿qué debe hacerse con las tramas correctas que le siguen? Se debe considerar que en el tipo de red en análisis, la capa de enlace de datos receptora está obligada a entregar paquetes a la capa de red en orden.

Además, en este protocolo también se ha omitido el supuesto de que la capa de red siempre tiene un suministro infinito de paquetes para enviar, por tanto cuando

la capa de red tiene un paquete para enviar a la capa de enlace de datos, causará la ocurrencia de un evento. Sin embargo, para poder cumplir con la regla de control de flujo, la capa de enlace de datos debe poder prohibir a la capa de red que la moleste con más trabajo.

1.5.2.2. Diseño e implementación en C++

Una manera para solventar el problema que surge cuando se pierde una trama, consiste en que el receptor simplemente descarte todas las tramas subsecuentes, sin enviar confirmaciones de recepción para las tramas descartadas.

Esta estrategia corresponde a una ventana de recepción de tamaño 1, en otras palabras, la capa de enlace de datos se niega a aceptar cualquier trama, excepto la siguiente que debe entregarse a la capa de red.

El emisor proveerá un búfer en el que almacenará los paquetes contenidos en las tramas transmitidas que aún no se hayan confirmado; si el búfer se llena antes de terminar el temporizador, el canal comenzará a vaciarse. En algún momento, el emisor terminará de esperar y retransmitirá en orden las tramas cuya confirmación no han llegado.

Al igual que en el protocolo de “ventana corrediza de un bit”, que implementa una transmisión bidireccional, se ha considerado implementar un solo módulo que cumpla con los roles de emisor y receptor; esto se lo ha realizado mediante la clase *Protocolo5*.

Debido a que este protocolo tiene múltiples tramas pendientes, necesita lógicamente múltiples temporizadores, uno por cada trama pendiente **[7].**

Los temporizadores se han implementado a través de las funciones que se utilizaron en los protocolos anteriores.

Para representar los eventos que envía la capa de red a la capa de enlace de datos en indicación de que tiene nuevos paquetes para enviar, también se han utilizado temporizadores.

A continuación, en la Figura 2.53 se presenta el diagrama de flujo para el protocolo de “ventana corrediza con retroceso N”.

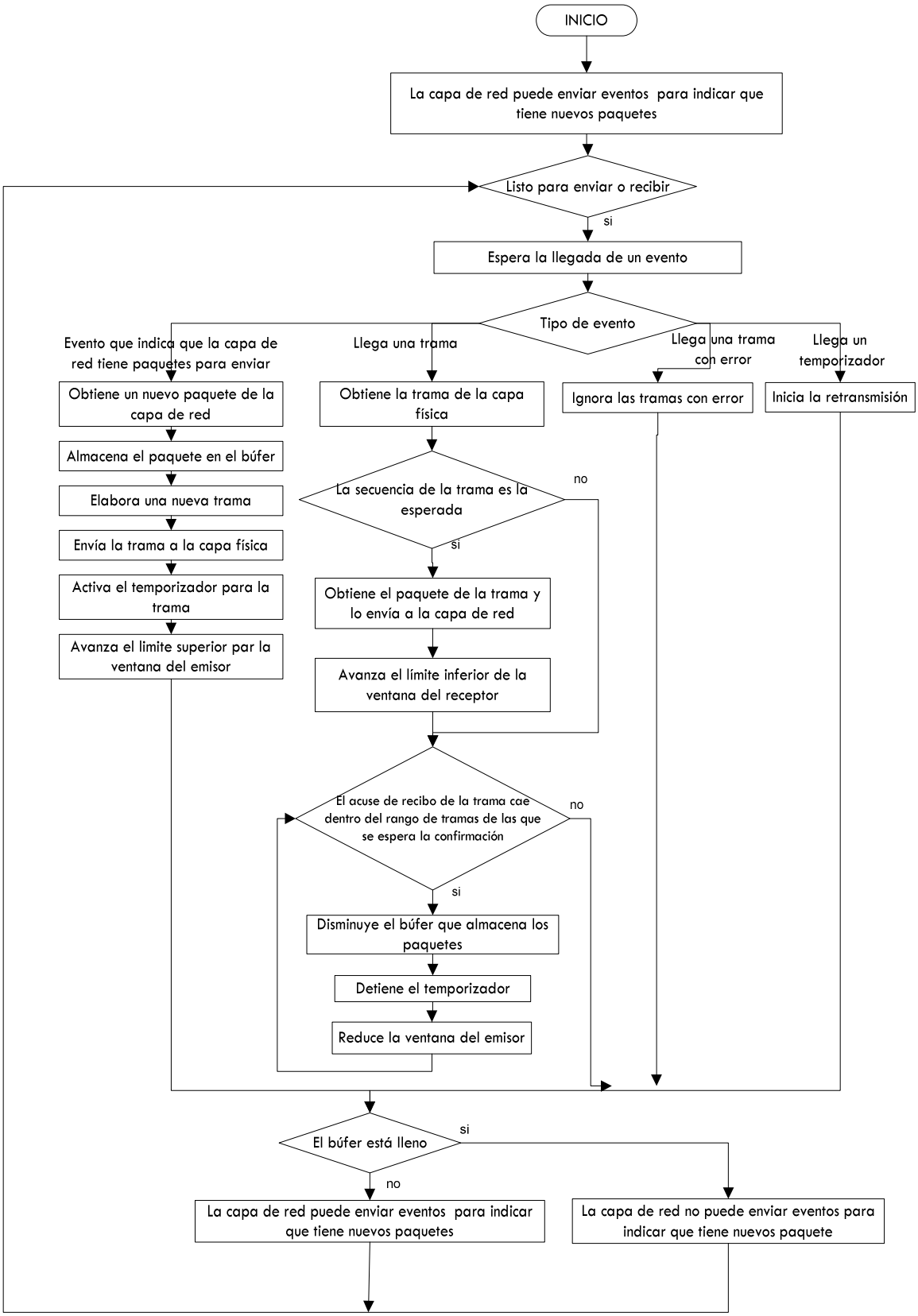


Figura 2.53 Diagrama de flujo para el protocolo de “ventana corrediza con retroceso N”.

1.5.2.2.1. Arreglos de temporizadores

Para la implementación de arreglos de temporizadores se ha utilizado objetos de la clase *cArray* que provee OMNET++. Un objeto de la clase *cArray* actúa como un contenedor de punteros a objetos de tipo *cObject* o de clases derivadas de ésta. Las principales funciones que se utilizan de esta clase son:

- **add():** Función que permite almacenar el objeto que se indica en su argumento dentro del arreglo y retorna la posición ocupada por éste.
- **addAt():** Función que permite almacenar el objeto especificado en su primer argumento dentro del arreglo de acuerdo a una posición dada en su segundo argumento; si la posición está siendo ocupada, la función lanza una excepción indicando esa situación.
- **remove():** Remueve los objetos almacenados en el arreglo; en su argumento se puede indicar que objeto se desea remover dándole el índice, nombre o el puntero al objeto.
- **exist():** Función que permite verificar si la posición que se indica en su argumento se encuentra ocupada o libre.
- **takeOwnership():** De acuerdo a su argumento (*true* o *false*) se determina si el contenedor tomará el control total sobre los objetos que se inserten en él; es así que si el argumento es *false* evitará que al destruir el contenedor se destruyan los objetos contenidos en él.

1.5.2.2.2. Temporizadores

Como se mencionó anteriormente, en este protocolo se requieren dos tipos de temporizadores, representados por objetos de la clase *cMessage* y *frame*:

- **event_timeout:** Puntero a *frame* que representa un temporizador asociado a cada trama que se envía, su definición e implementación es similar a los temporizadores descritos en los protocolos anteriores, por tanto, no se profundizará en su explicación.

  Se crearán copias de este objeto para insertarlas en el arreglo de temporizadores (*arrayTimer*).

- **event_network_layer_ready:** Puntero a *cMessage* que representa un evento que envía la capa de red a la capa de enlace de datos en comunicación de que tiene nuevos paquetes; esto se lo implementa en la función *enable_network_layer().*

  Se crearán copias de este objeto para insertarlas en el arreglo de eventos (*arrayNetworkLayerReady*).

1.5.2.2.3. Módulo

Clase Protocolo5

El objetivo de esta clase es proveer funcionalidad para:

- Proveer funciones que indiquen a la capa de red cuando está lista o no para recibir nuevos paquetes.
- Generar y esperar (en orden) la llegada de tramas de datos y de control.
- Iniciar un temporizador por cada una de las tramas enviadas y retransmitir la trama para la cual el temporizador expire y todas las consecutivas a ésta.

En el Código Protocolo 5.1 se presenta la definición de la clase *Protocolo5Agent.*

```
#define MAX_SEQ 7
typedef int seq_nr;

typedef enum{FRAME=1,TIMEOUT=2,READY=3,ACKTIMEOUT=4} tipo_evento;

class Protocolo5: public cSimpleModule
{
private:
// Variables del protocolo

     seq_nr next_frame_to_send;    // Variable que representa
                                   // el número de secuencia de
                                   // la siguiente trama de salida
     seq_nr ack_expected;          // Variable que representa la trama
                                   // más antigua hasta el momento no
                                   // confirmada
     seq_nr frame_expected;        // Variable que representa
                                   // el número de secuencia de
                                   // la trama que se espera recibir
     frame *s,*r;                  // Variables que representan a las
                                   // tramas que serán enviadas y
                                   // recibidas respectivamente
```

```
    packet buffer[MAX_SEQ+1];     // Búfer para el flujo de salida
    seq_nr nbuffered;             // Número de búferes de salida
                                  // actualmente en uso
    int frameSize;                // Tamaño de la trama
    int i_event;                  // Contador de eventos
                                  // network_layer_ready
    double vtx;                   // Velocidad de transmisión del canal
    double loss;                  // Variable que representa la tasa de
                                  // pérdida en el canal considerando
                                  // el valor de channel_behavior_loss

    double err;                   // Variable que representa la tasa de
    // error en el canal considerando
    // el valor de channel_behavior_err


    double channel_behavior_err;  // Variable que representa el
                                  // comportamiento del canal a través
                                  // de una distribución de
                                  // probabilidad, en base a esta se
                                  // conoce cuando se generan
                                  // tramas erradas


    double channel_behavior_loss;// Variable que representa el
                                  // comportamiento del canal a través
                                  // de una distribución de
                                  // probabilidad, en base a esta se
                                  // conoce cuando se generan
                                  // pérdidas

    const char* case_;            // Variable que permite seleccionar
                                  // el módulo iniciará la transmisión


    // Variables para los temporizadores
    double timer;           // Tiempo en el que expirará el
                            // temporizador
    double TimeEventBefore; // Almacena el tiempo programado
                            // para que el evento de tipo
                            // NetworkLayerReady sea consumido

    cArray *arrayNetworkLayerReady; // Arreglo de eventos
                                    // event_network_layer_ready
    cMessage *event_network_layer_ready;// Variable que representa al
                            // evento event_network_layer_ready del
                            // cual se tomará una copia y que será
                            // almacenado en el arreglo
                            // arrayNetworkLayerReady

    cArray *arrayTimer;     // Arreglo de eventos event_timeout
    frame *event_timeout;   // Variable que representa al temporizador
                            // del cual se tomará una copia y se
                            // insertará en el arreglo arrayTimer

public:
// Constructor
    Protocolo5();
```

```
// Destructor
     virtual ~Protocolo5();

// Funciones propias del protocolo
     void from_network_layer(packet* msg);     // Obtiene un paquete
                                               // de la capa de red
     void to_network_layer(packet *msg);       // Envía un paquete a
                                               // la capa de red
     void to_physical_layer(frame * envio);    // Envía una trama
                                               // hacia la capa física

     bool between(seq_nr a,seq_nr b,seq_nr c);// Función que permite
                           // verificar si los valores recibidos caen
                            // dentro del rango de valores esperados

     // Elabora y envía una trama de datos
     void send_data(seq_nr frame_nr,seq_nr frame_expected,
     packet buffer[]);

// Funciones para los temporizadores
     void start_timer(seq_nr k);   // k hace referencia al temporizador
                                   // que corresponde a la trama de un
                                   // número de secuencia dado
     void stop_timer(seq_nr k);    // k hace referencia al ack_expected

     void enable_network_layer();  // Función que permite indicar a la
                                   // capa de red que puede enviar
                                  // nuevos paquetes
     void disable_network_layer(); // Función que indica a la capa de
                                   // red que no puede recibir más
                                  // paquetes

// Funciones auxiliares
     void inc(int& k);             // Incrementa circularmente
                                   // el valor de la trama que se está
                                   // esperando enviar o recibir
     void crear_paquete(packet *msg);// Genera un paquete en la
                                     // capa de red


     void case_network_layer_ready();    // Función que es invocada
                                         // cuando llega un evento
                                   // event_network_layer_ready

     void case_timeout();// Retransmite una trama cuando
                         // se expira el temporizador
     void remueve_evento(cMessage *msg);// Remueve un objeto del arreglo
                                        // arrayNetworkLayerReady
     void registro_evento(); // Imprimir en pantalla el nombre del
                             // módulo y el tiempo en que ocurre un
                             // evento en éste
protected:
// Funciones de la clase cSimpleModule que se redefinen
    virtual void initialize();     // Función que es invocada para
                                   // iniciar la simulación
     virtual void handleMessage(cMessage *p_r_);
};
```

Código Protocolo 5.1 Definición de la clase *Protocolo5*.

Variables

- **seq_nr ack_expected**: Variable que representa la trama más antigua, hasta el momento, no confirmada.
- **nbuffered:** Representa el número de búferes de salida actualmente en uso.
- **i_event:** Representa el identificador del último evento que se ha enviado a la capa de red; su valor se incrementará de manera circular de acuerdo al tamaño de la ventana.
- **buffer[MAX_SEQ + 1]**: Arreglo estático que almacena los paquetes obtenidos de la capa de red cuyo tamaño depende del valor de la ventana, ya que en caso de que se pierdan o dañen, las tramas que se encuentran dentro de la ventana del emisor, se podrán recuperar los paquetes para generar nuevas tramas y retransmitirlas.
- **arrayNetworkLayerReady:** Arreglo que contiene punteros a copias de *event_network_layer_ready*.
- **arrayTimer:** Arreglo que contiene punteros a copias de *event_timeout*.
- **err:** Variable que representa la tasa de tramas con error.
- **channel_behavior_err:** Variable que representa el comportamiento del canal a través de una distribución de probabilidad para la generación de tramas con error.

Funciones

- **initialize():** Función que será invocada para empezar con la transmisión de datos. Dentro de ella se invocará a la función *enable_network_layer(),* además se inicializan los punteros de la clase *cArray* que servirán para almacenar los temporizadores y eventos que habilitan a la capa de red.

  La implementación de la función *initialize()* se presenta a continuación en el Código Protocolo 5.2.

```
void Protocolo5::initialize()
{
     s=new frame("frame",FRAME);
     // Inicialización de variables del protocolo
     next_frame_to_send=0;
     frame_expected=0;
     nbuffered=0;
     ack_expected=0;
     i=0;
     i_event=0;

     // Se toman los valores configurados en el archivo de
     // inicialización .ini o en el archivo .ned
     timer= par("timer");
     frameSize= par ("frameSize");
     vtx= par("vtx");
     case_ =  (const char *)par("case_");
     loss = par("loss");
     err= par(err);

     /* -- Arreglo de eventos event_network_layer_ready --*/
     // Inicialización del arreglo de eventos event_network_layer_ready
     arrayNetworkLayerReady=new cArray("event_network_layer_ready",8);

     // Evita que el contenedor tome el control total
     // sobre los objetos que se inserten en él;
     arrayNetworkLayerReady->takeOwnership(false);

     // Se da el nombre y el tipo al puntero event_network_layer_ready
     event_network_layer_ready=new cMessage("NetworkLayerReady",READY);

     /*-- Arreglo de eventos que representan a los temporizadores --*/
     // Inicialización del Array de eventos temporizadores
     arrayTimer=new cArray("event_timeout");

     // Evita que el contenedor tome el control total
     // sobre los objetos que se inserten en él;    arrayTimer-
>takeOwnership(false);

     // Se da el nombre y el tipo al puntero event_timeout
     event_timeout=new frame("timeout",TIMEOUT);

     // Se configura la variable que será utilizada para
     // la generación de los eventos event_network_layer_ready
     TimeEventBefore=simTime();

     // Permite seleccionar el módulo que empezará con la transmisión
     if((strcmp(case_, name()) == 0))
     {
          enable_network_layer();
     }
}
```

Código Protocolo 5.2 Implementación de la función *initialize ()* de la clase *Protocolo5.*

- **between():** Función que será utilizada para verificar si los valores recibidos caen dentro del rango de valores esperados. En la

implementación de este protocolo, la función *between()* es utilizada para confirmar si el acuse de recibo coincide con el valor de la siguiente trama que se va a enviar; adicionalmente, se confirma implícitamente a todas las tramas que aún no han sido confirmadas. En el Código Protocolo 5.3 se indica la implementación de esta función.

```
bool Protocolo5::between(seq_nr a,seq_nr b,seq_nr c)
{
     if(((a<=b)&&(b<c)) || ((c<a)&&(a<=b)) || ((b<c)&&(c<a)))
          return (true);
     else
          return (false);
}
```

Código Protocolo 5.3 Implementación de la función *between()* de la clase *Protocolo5*.

- **send_data():** Función en la que se forma una trama (se asignan los valores a los campos) y se la envía a la capa física. Su implementación se la presenta en el Código Protocolo 5.4.

```
void Protocolo5::send_data(seq_nr frame_nr,seq_nr frame_expected,packet
buffer[])
{
     // Se coloca el paquete en el campo info de frame
     s->setInfo(buffer[frame_nr]);
     // Se inserta el número de secuencia en campo seq de frame
     s->setSeq(frame_nr);
     // Se inserta el número de ack en campo ack de frame
     s->setAck((frame_expected + MAX_SEQ)%(MAX_SEQ+1));
     // Se asigna el tamaño de la trama de acuerdo al valor
     // configurado en el archivo .ini o .ned
     s->setLength(frameSize);
     // Se crea una copia para enviar
     frame *copia=(frame*)s->dup();
     to_physical_layer(copia);
     start_timer(frame_nr);
}
```

Código Protocolo 5.4 Implementación de la función *send_data()* de la clase *Protocolo5*.

- **enable_network_layer():** Función que es utilizada para indicar a la capa de red que puede recibir un nuevo paquete; esta será invocada cada vez que se haga la verificación del búfer y éste no se encuentre lleno; en esta función se reprogramará un temporizador (*event_network_layer_ready*) que expirará dependiendo del valor de la variable *delay.*

  El valor de la variable *delay* es igual a un valor aleatorio más el tiempo de transmisión (*tx*) para evitar el encolamiento de las tramas;

adicionalmente, se hace una comprobación para programar eventos que se consuman progresivamente.

Para la generación del valor de la variable *delay*, se utiliza la función *intrand(n)* de OMNET++ que genera un valor aleatorio entre 0 y *n-1*. La implementación de la función *enable_network_layer()* se la presenta en el Código Protocolo 5.5.

```
void Protocolo5::enable_network_layer( )
{
     double tx;        // Variable que almacena el tiempo de transmisión
     double delay;     // Variable que indica el tiempo en que la capa
                       // de enlace de datos puede recibir un nuevo
                       // paquete
     do{
           tx=((double)frameSize)/(vtx);
     // Se configura un retardo que será entre 0.000 s y 0.0096 s
           delay=(intrand(16))*0.0001+tx;
     // Se hace un comprobación para que los eventos programados se
     // consuman progresivamente, adicionalmente se evita que exista
     // encolamiento
           if(simTime()+delay>TimeEventBefore &&
           (simTime()+delay-TimeEventBefore)>=tx )
            {
                 // Se crea un evento auxiliar para programarlo
                 cMessage *p_network_layer_ready_aux;
                 // Si no existen eventos libres para programarlos
                       if((arrayNetworkLayerReady
                          ->exist(i_event))==false)
                       {
           // Se almacena el tiempo en que se espera la llegada del
           // último evento event_network_layer_ready
                          TimeEventBefore=simTime()+delay;
           // Se copia el evento event_network_layer_ready
                          p_network_layer_ready_aux=(cMessage*)
                          event_network_layer_ready->dup();
           // Se agrega el temporizador al arreglo de
           // arrayNetworkLayerReady
                          arrayNetworkLayerReady->addAt
                         (i_event,p_network_layer_ready_aux);
           // Se programa al evento que será recibido luego de concluir
           // el tiempo determinado en la variable delay
                          scheduleAt(simTime()+delay,
                          p_network_layer_ready_aux);
                       }
                       // Se cambia el valor de i_event
                 inc(i_event);
           }
     }while((simTime()+delay)<TimeEventBefore && ((simTime()+delay)
     -TimeEventBefore)<tx );

}
```

Código Protocolo 5.5 Implementación de la función *enable_network_layer*() de la clase *Protocolo5*.

- **disable_network_layer():** Función que es utilizada para indicar a la capa de red que no envíe más paquetes a la capa de enlace de datos; por tanto no programa eventos *event_network_layer_ready*.
- **handleMessage():**Función que será invocada automáticamente por el *kernel* del simulador cuando llegue un objeto que represente a un evento de tipo: *READY, FRAME,* o *TIMEOUT*. El tipo de evento se encuentra determinado por el valor que retorna la función *kind()* de la clase *cMessage.*

  Si el evento recibido es de tipo *READY* (copia del puntero *event_network_layer_ready*) se invoca a la función *case_network_layer_ready(),* se remueve el puntero del arreglo y se lo elimina.

  Si el evento recibido es de tipo *FRAME* se convierte el puntero *p_r_* de tipo *cMessage* a tipo *frame y s*e procede a comprobar si éste está con error, para lo cual se utiliza la función *hasBitError(). S*i la trama está con error simplemente se la descarta, caso contrario se la acepta y se la procesa; se verifica si el número de secuencia de la trama recibida es la misma que la esperada, de ser así, se obtiene el paquete del campo de datos, se lo envía a la capa de red, y se avanza el límite inferior de la ventana del receptor. Además se obtiene el campo correspondiente al acuse de recibo, con el cual se determina que tramas llegaron correctamente para detener sus temporizadores y disminuir el búfer de paquetes enviados en las tramas.

  Si el evento recibido es de tipo *TIMEOUT*, se invoca a la función *case_timeout()* y se elimina el puntero.

  Finalmente, se verifica si el búfer está lleno, de ser así se invocará a la función *disable_network_layer(),* caso contrario se invocará a la función *enable_network_layer().*

La implementación de la función *handleMessage()* se la presenta en el Código Protocolo 5.6.

```
void Protocolo5::handleMessage(cMessage *p_r_)
{
     double tiempo=simTime();
     registro_evento();

     // Se obtiene el tipo de evento
     int event=p_r_->kind();
     switch (event)
     {
/*..................network_layer_ready..................*/
           case READY :
                 ev<<" Llega el evento "<<p_r_->name()<<endl;
                 remueve_evento(p_r_);
                 cancelEvent(p_r_);
                 delete p_r_;
                 case_network_layer_ready();
           break;
/*..................frame_arrival..................*/
           case FRAME:
                 r = check_and_cast<frame*>(p_r_);
                 // Se verifica si la trama está con error
                 if(r->hasBitError()==true)
                       {
                             ev<<" Llegó una trama con error"<<endl;
                             bubble("LLegó trama con error");
                             delete p_r_;
                       }


                else
                 {
                       ev<<" Recibe la trama (seq,ack) : ("
                       <<r->getSeq()<<","<<r->getAck()<<")"<<endl;
                       if(r->getSeq()==frame_expected)
                       {
                             // Envía el paquete a la capa de red
                             to_network_layer(&r->getInfo());
                             // Se cambia el valor de la trama esperada
                             inc(frame_expected);
                       }
           /*
           Se confirma si el acuse de recibo cae dentro del rango de
           número de secuencia de las tramas de las que se espera la
           confirmación, además se confirma implícitamente la recepción
           de las tramas anteriores
           */
                       while (between(ack_expected,
                       r->getAck(),next_frame_to_send))
                       {
                       // Indica que hay una trama menos en el búfer
                             nbuffered=nbuffered-1;
                             // Se detiene al temporizador
                             stop_timer(ack_expected);
                             // Se cambia el valor del ack esperado
                             inc(ack_expected);
                       }
                 // Se borra la trama recibida
                 delete p_r_;
                 }
           break;
```

```
    /*.................timeout.................*/
          case TIMEOUT :
                case_timeout();

          break;
          }

          ev<<" El tamaño del búfer es: "<<nbuffered<<endl;

          // Si el búfer aún no está lleno, se le comunica a la capa
          // de red para que pueda enviar mas paquetes
          if(nbuffered<MAX_SEQ)
          {
                enable_network_layer();
          }
          else
          // Si el búfer está lleno, se le comunica a la capa
          // de red que no envíe mas paquetes
                disable_network_layer();
}
```

Código Protocolo 5.6 Implementación de la funcion *handleMessage()* de la clase *Protocolo5*.

- **case_network_layer_ready():** Función que será invocada cuando llegue un evento *event_network_layer_ready*; en esta función se verifica si el búfer está lleno, de ser así no se permite programar más eventos *event_network_layer_ready*, caso contrario se obtiene un nuevo paquete de la capa de red, se incrementa el tamaño de búfer, se crea y envía una nueva trama, y se expande la ventana del emisor.

En el Código Protocolo 5.7 se presenta su implementación.

```
void Protocolo5::case_network_layer_ready()
{
     if(nbuffered<7)
     {
          from_network_layer(&buffer[next_frame_to_send]);
          nbuffered++;
          send_data(next_frame_to_send,frame_expected,buffer);
          inc(next_frame_to_send);
     }
       else
     {
          disable_network_layer();
     }
}
```

Código Protocolo 5.7 Implementación de la función *case_network_layer_ready()* de la clase *Protocolo5.*

- **case_timeout():** Función que será invocada cuando un temporizador programado expire para proceder a reenviar todas las tramas desde

aquella para la cual expiró el temporizador hasta la última que se envió. En el Código Protocolo 5.8 se presenta su implementación.

```
void Protocolo5::case_timeout()
{
     // Se asigna el valor de la trama desde la cual
     // se expiró el timer para reenviar
     next_frame_to_send=ack_expected;
     ev<<" !!!! Temporizador expirado para la trama
     "<<next_frame_to_send<<endl;
     bubble("Temporizador expirado");
     for(int i =1 ;i<=nbuffered; i++)
     {
           // Detiene el temporizador
           stop_timer(next_frame_to_send);
           // Se procede el entramado de las tramas a retransmitir
           send_data(next_frame_to_send,frame_expected,buffer);
           // Se cambia el valor de next_frame_to_send
           inc(next_frame_to_send);
     }
}
```

Código Protocolo 5.8 Implementación de la función *case_timeout ()* de la clase *Protocolo5*.

A diferencia de los protocolos anteriores, en éste se ha configurado tanto un modelo de error como uno de pérdida. El modelo de error se ha implementado de forma similar que el modelo de pérdida en la función *to_physical_layer()* pero la verificación de enviar o no la trama con error se hará con la variable *err* que representa la tasa de tramas con error. En el Código Protocolo 5.9 se presenta la implementación.

```
void Protocolo5::to_physical_layer(frame *envio)
{
     // Se asigna a la variable channel_behavior_err
     // y a la variable channel_behavior_loss el
     // valor generado en el archivo omnetopp.ini
     channel_behavior_err=par("channel_behavior_err");
     channel_behavior_loss=par("channel_behavior_loss");

     // Si se cumple la condición de que el valor generado para la
     // variable channel_behavior es menor al valor err, entonces
     // se configura como el valor de true a la variable error de
     // cMessage

     if(channel_behavior_err<err)
     {
           ev<<" Trama con error "<<endl;
           // Se muestra en la simulación el texto
           // Trama con error
           bubble(" Trama con error");
           // Se configura la variable error de la trama que se envía
           ev<<" Envía la trama (seq,ack) : ("
            <<envio->getSeq()<<","<<envio->getAck()<<")"<<endl;
```

```
            envio->setBitError(true);
            send(envio,"salida");

      }

      // Si se cumple la condición de que el valor generado para la
      // variable channel_behavior es menor al asignado a loss, entonces
      // se elimina la trama
      else if (channel_behavior_loss<loss)
      {
            ev<<" Pierde la trama "<<endl;
            // Se muestra en la simulación el texto
            // Trama Perdida
            bubble(" Trama Perdida");
            // Se borra la trama
            delete envio;
      }
      else
      {
            // Envía una copia de la trama que recibe desde la capa de
            // enlace
            ev<<" Envía la trama (seq,ack) : ("
             <<envio->getSeq()<<","<<envio->getAck()<<")"<<endl;
            send(envio,"salida");
       }
}
```

Código Protocolo 5.9 Implementación del modelo de pérdida y error de tramas en la función *to_physical_layer()* del módulo.

1.5.2.3. Configuración del escenario de simulación

1.5.2.3.1. Configuración del escenario de simulación en el lenguaje NED

```
// Definición del módulo emisor y receptor
simple Protocolo5

    parameters:
    frameSize:numeric, // Tamaño de la trama
    vtx:numeric,       // Velocidad de transmisión
    channel_behavior_loss:numeric,// Comportamiento del canal para
                                  // pérdidas de tramas
    channel_behavior_err:numeric, // Comportamiento del canal para
                                  // errores en las tramas
    loss:numeric,      // Tasa de pérdida de tramas
    case_:string,      // Permite escoger el módulo que empieza
                       // la transmisión
    err:numeric,       // Tasa de tramas erradas
    timer:numeric;     // Tiempo en que expirará el temporizador

    gates:
        out: salida;   // Compuerta de salida
        in: entrada;   // Compuerta de entrada

endsimple
```

```
module Protocolo5_

     parameters:
           frameSize:numeric,
           vtx:numeric,
           channel_behavior_loss:numeric,
           channel_behavior_err:numeric,
           loss:numeric,
           case_:string,
           err:numeric,
           timer:numeric,
           propagacion:numeric;
    submodules:
        A: Protocolo5;  // A: Instancia del módulo
                        // Protocolo5
           parameters:
                 frameSize=frameSize,
                 vtx=vtx,
                 channel_behavior_loss=channel_behavior_loss,
                 channel_behavior_err:channel_behavior_err,
                  loss=loss,
                  err=err,
                  case_=case_,
                 timer=timer;
            display: "p=35,96;i=device/pc,cyan";
        B: Protocolo5; // B: Instancia del módulo
                       // Protocolo5
        parameters:
                 frameSize=frameSize,
                 vtx=vtx,
                 channel_behavior_loss=channel_behavior_loss,
                 channel_behavior_err:channel_behavior_err,
                  loss=loss,
                  err=err,
                  case_=case_,
                 timer=timer;
           display: "p=224,96;i=device/pc,gold";
    connections:
        A.salida --> delay propagacion datarate vtx --> B.entrada;
        B.salida --> delay propagacion datarate vtx --> A.entrada;
        display: "b=250,250";
endmodule

network Network : Protocolo5_
endnetwork
```

Escenario Protocolo5 Configuración del escenario para simular el protocolo de “ventana corrediza con retroceso N”.

1.5.2.3.2. Configuración de los parámetros en el archivo omnetpp.ini

Adicionalmente a la configuración en el archivo .ini del protocolo anterior, en éste se configura la tasa de tramas erradas (*err*).

En este protocolo solo un módulo empezará con la transmisión.

```
[General]
preload-ned-files=protocolo5.ned
network=Network
sim-time-limit = 1s
[Parameters]
Network.frameSize=8000
Network.vtx=1000000          # Velocidad de transmisión en bits por s
Network.timer=0.100          # Tiempo para que el temporizador expire
Network.propagacion=0.02     # Tiempo de propagación en el canal en s
Network.loss=0.01            # Tasa de pérdida de tramas
Network.err=0.03             # Tasa de tramas con error
Network.case_="A"            # Módulo que empieza la transmisión
Network.channel_behavior_err=uniform(0,1)
Network.channel_behavior_loss=uniform(0,1)

[Run 0]
description="Protocolo5"
[Run 1]
description="Protocolo5, empieza A"
Network.case_="A"
[Run 2]
description="Protocolo5, empieza B"
Network.case_="B"
```

Figura 2.54 Configuración de los parámetros en omnetpp.ini para el protocolo de "ventana corrediza con retroceso a N".

1.5.2.4. Resultados de la simulación

1.5.2.4.1. Impresión en pantalla

A continuación, en la Figura 2.55 se indican algunos de los resultados que se imprimen en la pantalla al realizar la simulación del protocolo "ventana corrediza con retroceso N"

```
. ** Event #0.  T=0.0092000000 (  9ms).  Module #2 `Network.A'
---------------------------------------------
NODO:A, TIME:9.2 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (seq,ack): (0,7)
 Iniciando el temporizador 0
 El tamaño del búfer es: 1
** Event #1.  T=0.0172000000 ( 17ms).  Module #2 `Network.A'
---------------------------------------------
NODO:A, TIME:17.2 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (seq,ack) :(1,7)
 Iniciando el temporizador 1
 El tamaño del búfer es: 2
.
.
.
```

```
NODO:B, TIME:37.2 ms
 Recibe la trama (seq,ack):(0,7)
 La capa de red recibe:Nuevo Paquete
 El tamaño del búfer es: 0
** Event #5.  T=0.0422000000 ( 42ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:42.2 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (seq,ack) :(4,7)
 Iniciando el temporizador 4
 El tamaño del búfer es: 5
** Event #6.  T=0.0452000000 ( 45ms).  Module #3 `Network.B'
-----------------------------------------------
NODO:B, TIME:45.2 ms
 Recibe la trama (seq,ack):(1,7)
 La capa de red recibe:Nuevo Paquete
 El tamaño del búfer es: 0
.
.
.
** Event #162.  T=0.4567000000 (456ms).  Module #3 `Network.B'
-----------------------------------------------
NODO:B, TIME:456.7 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Trama con error
 Envía la trama (seq,ack) : (0,0)
 Iniciando el temporizador de la trama 0
 El tamaño del búfer es: 6.
.
.
** Event #173.  T=0.4847000000 (484ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:484.7 ms
 Llegó una trama con error
 Recibe la trama (seq,ack) : (0,0)
 El tamaño del búfer es: 7
 Búfer lleno, no se generan eventos.
.
** Event #198.  T=0.5567000000 (556ms).  Module #3 `Network.B'
-----------------------------------------------
NODO:B, TIME:556.7 ms
 !!!! Temporizador expirado para la trama 0
 Detiene el temporizador :0
 Trama con error
 Envía la trama (seq,ack) : (0,1)
 Iniciando el temporizador de la trama 0
 Detiene el temporizador :1
 Envía la trama (seq,ack) : (1,1)
 Iniciando el temporizador de la trama 1
 Detiene el temporizador :2
 Envía la trama (seq,ack) : (2,1)
 Iniciando el temporizador de la trama 2
 Detiene el temporizador :3
 Envía la trama (seq,ack) : (3,1)
 Iniciando el temporizador de la trama 3
 Detiene el temporizador :4
 Envía la trama (seq,ack) : (4,1)
 Iniciando el temporizador de la trama 4
```

```
 Detiene el temporizador :5
 Envía la trama (seq,ack) : (5,1)
 Iniciando el temporizador de la trama 5
 Detiene el temporizador :6
 Envía la trama (seq,ack) : (6,1)
 Iniciando el temporizador de la trama 6
 El tamaño del búfer es: 7
 Búfer lleno, no se generan eventos
** Event #214.  T=0.6310000000 (631ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:631 ms
 !!!! Temporizador expirado para la trama 2
 Detiene el temporizador :2
 Envía la trama (seq,ack) : (2,7)
 Iniciando el temporizador de la trama 2
 Detiene el temporizador :3
 Envía la trama (seq,ack) : (3,7)
 Iniciando el temporizador de la trama 3
 Detiene el temporizador :4
 Pierde la trama
 Iniciando el temporizador de la trama 4
 Detiene el temporizador :5
 Envía la trama (seq,ack) : (5,7)
 Iniciando el temporizador de la trama 5
 Detiene el temporizador :6
 Envía la trama (seq,ack) : (6,7)
 Iniciando el temporizador de la trama 6
 Detiene el temporizador :7
 Envía la trama (seq,ack) : (7,7)
 Iniciando el temporizador de la trama 7
 Detiene el temporizador :0
 Envía la trama (seq,ack) : (0,7)
 Iniciando el temporizador de la trama 0
 El tamaño del búfer es: 7
 Búfer lleno, no se generan eventos
```

Figura 2.55 Impresión en pantalla obtenida de la simulación del protocolo “ventana corrediza con retroceso N”.

En los resultados que se presentan en la Figura 2.55 se puede apreciar:

- Cuando un módulo envía correctamente una trama y ésta a su vez es recibida (por ejemplo al tiempo 9.2ms el módulo A envía la trama (0,7) y ésta es recibida por el módulo B al tiempo 37.2ms).
- Cuando una trama que es enviada desde un módulo es marcada con error y al ser descartada por el receptor no tiene confirmación, por lo que expira el temporizador correspondiente (por ejemplo al tiempo 456.7ms la trama que es enviada por el módulo B es marcada con error y al tiempo 556.7ms expira el temporizador correspondiente).

- Cuando se pierde una trama que es enviada por un módulo (por ejemplo al tiempo 631ms se pierde una trama que es enviada desde el módulo A justamente cuando está realizando una retransmisión).

1.5.2.4.2. Visualización gráfica de la simulación

A continuación, desde la Figura 2.56 hasta la Figura 2.61 se presentan los resultados que se visualizan con la simulación del protocolo “ventana corrediza con retroceso N”.

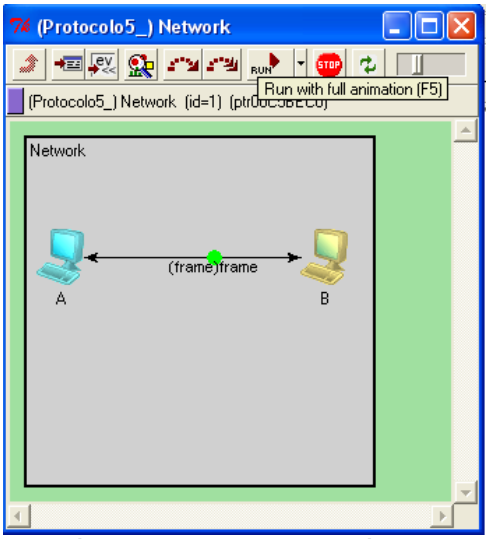


Figura 2.56 Envío de una trama del módulo A al módulo B.

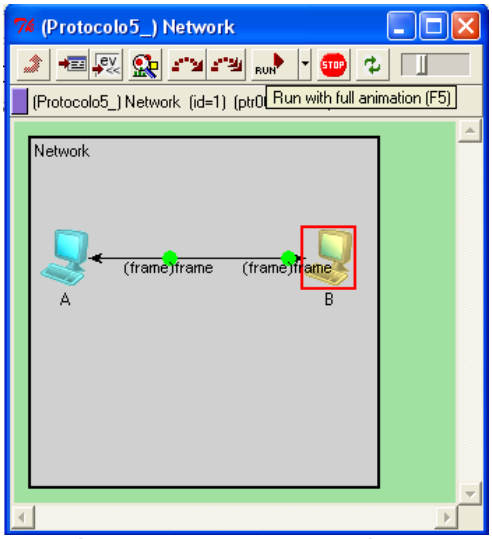


Figura 2.57 Envío de una trama del módulo B al módulo A.

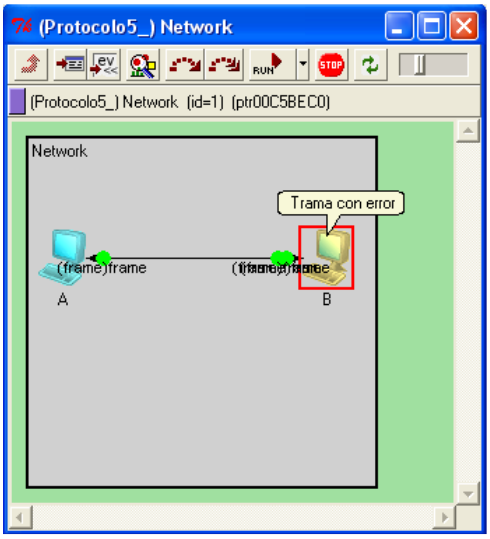


Figura 2.58 La trama a ser enviada es marcada con error.

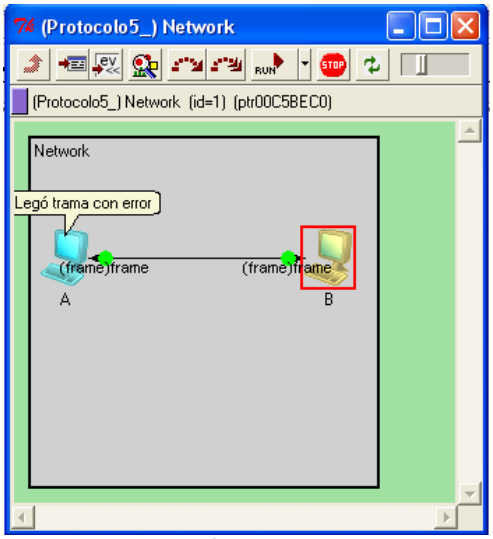


Figura 2.59 Detección de una trama con error.

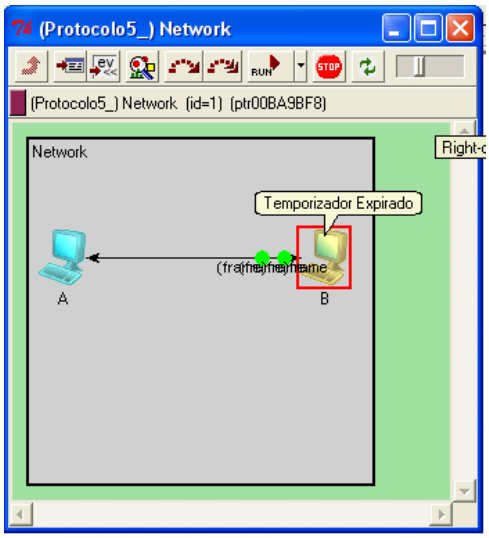


Figura 2.60 Temporizador expirado.

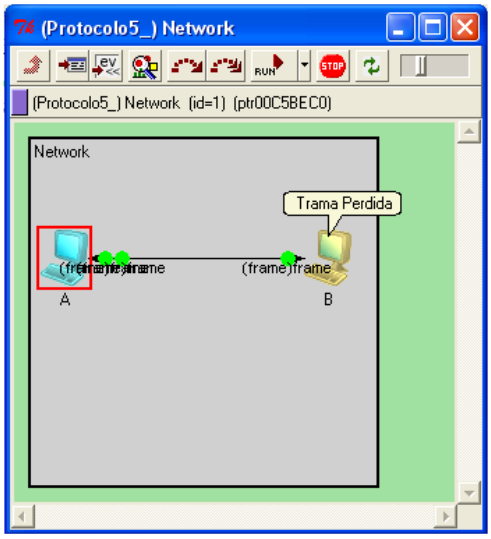


Figura 2.61 La trama a ser enviada se pierde.

Con la ejecución del protocolo se ha obtenido resultados con los que se verifica que están acorde a las consideraciones descritas para el mismo, es así que se puede observar que en comparación con los protocolos anteriores, en éste se aprovecha la utilización del canal de comunicación ya que no permanece desocupado mientras espera por la confirmación de las tramas enviadas, sin embargo se puede apreciar también la desventaja que presenta el protocolo al tener que reenviar todas las tramas posteriores a una trama cuyo temporizador ha expirado.

1.5.3. PROTOCOLO DE VENTANA CORREDIZA DE REPETICIÓN SELECTIVA

1.5.3.1. Descripción del protocolo

Este protocolo soluciona el problema que surge en el protocolo de “ventana corrediza con retroceso N” cuando los errores en el canal de comunicación son frecuentes, por tanto una estrategia del manejo de errores es permitir que el receptor acepte y coloque en búferes las tramas que siguen a una trama dañada o perdida y que llegan correctamente.

Las consideraciones para este protocolo son:

- Si la trama que llega tiene el número de secuencia esperado,se obtiene el paquete de ella y se lo pasa a la capa de red. Si llega una trama con un

número de secuencia diferente al esperado, ésta es almacenada en un búfer hasta tener todas las tramas en orden y poder pasarlas a la capa de red.

- El receptor tiene la capacidad de enviar una trama de confirmación de recepción negativa NAK cuando recibe una trama diferente a la que espera (esto ocurre cuando existen tramas perdidas o errores en el canal).
- El receptor puede enviar una trama de confirmación independiente en el caso de no existir una trama de datos para superponer dicha confirmación.
- Al igual que el protocolo de “ventana corrediza con retroceso N”, la capa de enlace de datos tiene la capacidad de bloquearse y no recibir nuevos paquetes de la capa de red cuando el búfer esté lleno.

La recepción no secuencial introduce ciertos problemas que no se presentan en los protocolos en los que las tramas sólo se aceptan en orden (ver la Figura 2.58).

Suponiendo que se tiene un número de secuencia de tres bits, se permite al emisor enviar hasta siete tramas antes de que se le exija que espere una confirmación de recepción. Inicialmente, las ventanas del emisor y del receptor están como se muestra en la Figura 2.62. El emisor ahora transmite las tramas desde la 0 hasta la 6. La ventana del receptor le permite aceptar cualquier trama con un número de secuencia entre 0 y 6, inclusive. Las siete tramas llegan correctamente, por lo que el receptor confirma su recepción y avanza su ventana para permitir la recepción de 7, 0, 1, 2, 3, 4 o 5. Los siete búferes se marcan como vacíos.

En este punto surgen problemas en el canal de comunicación si se pierden todas las confirmaciones de recepción. En algún momento termina el temporizador del emisor y retransmite la trama 0. Cuando esta trama llega al receptor, se efectúa una verificación para saber si está dentro de la ventana del receptor. Desgraciadamente, como se indica en la Figura 2.62, la trama 0 está dentro de la nueva ventana, por lo que se acepta, se obtiene su paquete y se almacena en el búfer de entrada. El receptor envía una confirmación de recepción superpuesta

para la última trama que se recibió correctamente en este caso la trama 6, y por tanto se confirma implícitamente que se ha recibido correctamente las tramas anteriores a ésta (desde la 0 hasta la 5).

El emisor se entera con beneplácito que todas sus tramas transmitidas han llegado de manera correcta, por lo que avanza su ventana y envía de inmediato las tramas 7, 0, 1, 2, 3, 4 y 5. El receptor aceptará la trama 7 y el paquete de ésta se pasará directamente a la capa de red. Inmediatamente después, la capa de enlace de datos receptora revisa si ya tiene una trama 0 válida, descubre que sí y pasa el paquete que contiene a la capa de red. En consecuencia, la capa de red obtiene un paquete incorrecto, y falla el protocolo.

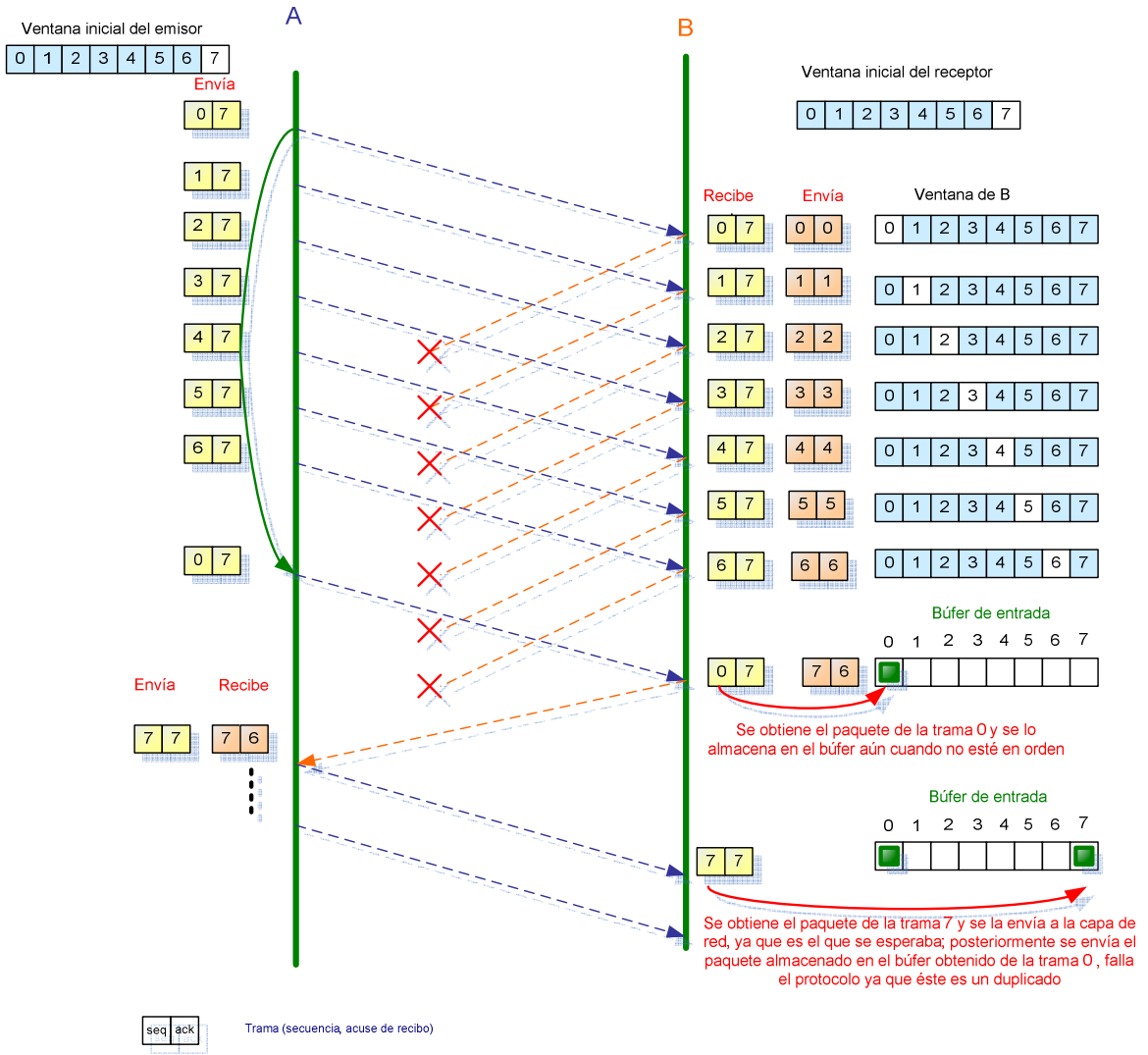


Figura 2.62 Problema que se presenta en el protocolo de ventana corrediza de repetición selectiva.

La esencia del problema ocurre cuando el receptor avanza su ventana y el nuevo intervalo de números de secuencia válidos se traslapa con el anterior. En consecuencia, el siguiente grupo de tramas podrían ser duplicadas (si se perdieron todas las confirmaciones de recepción) o nuevas (si se recibieron todas las confirmaciones de recepción) y el receptor no tiene manera de distinguirlas.

Para dar solución al problema descrito en el protocolo de “ventana corrediza de repetición selectiva” y evitar que haya traslape, el tamaño de la ventana deberá ser de al menos la mitad del intervalo de los números de secuencia; por tanto, para números de secuencia de 3 bits, el tamaño de ventana será 4.

1.5.3.2. Diseño e implementación en C++

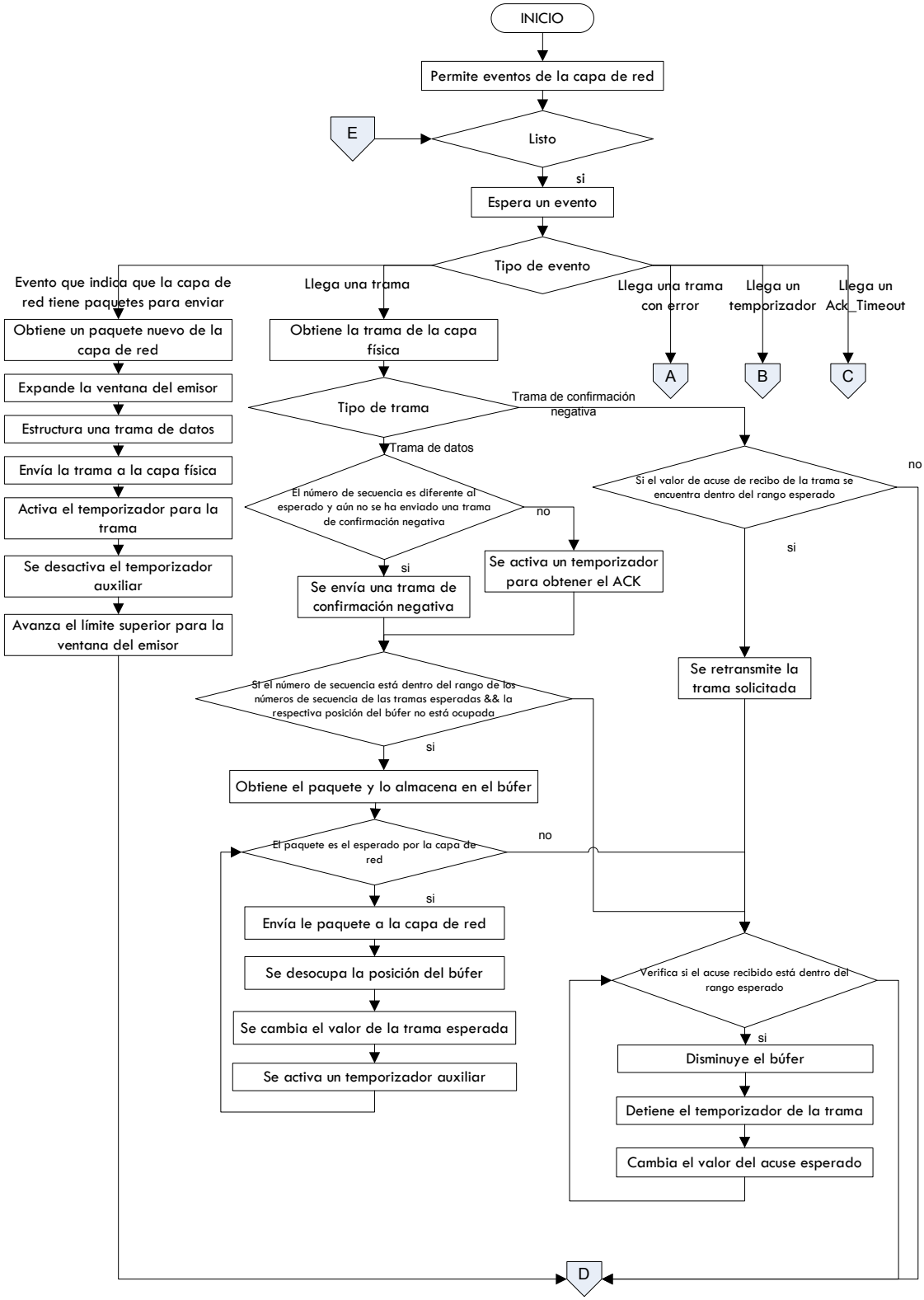
El número de búferes necesarios para almacenar los paquetes enviados en las tramas será igual al tamaño de la ventana, no al intervalo de números de secuencia, ya que en ninguna circunstancia puede aceptar tramas cuyos números de secuencia estén por debajo del extremo inferior o por encima del extremo superior de la ventana. Por la misma razón, el número de temporizadores requeridos es igual al número de búferes, no al tamaño del espacio de números secuencia; por tanto, hay un temporizador asociado a cada búfer. Cuando el temporizador expire, el contenido del búfer se retransmitirá.

Para la retransmisión del paquete contenido en el búfer, al cual está asociado el temporizador, se ha utilizado la variable *seq_var*.

En este protocolo, en caso de que no haya tráfico al cual superponer la confirmación de recepción, se inicializa un temporizador auxiliar (representado por el puntero *event_acktimeout* de tipo *frame*) tras la llegada de una trama de datos en secuencia, si éste expira antes de que se presente tráfico en la dirección contraria, se envía una trama de confirmación de recepción independiente. Esto evitará una retransmisión por expiración del temporizador asociado a la trama.

Al igual que en los dos protocolos anteriores, se ha implementado un solo módulo que cumpla con los roles de emisor y receptor, esto se lo ha realizado mediante la clase *Protocolo6*.

En la Figura 2.63 se presenta el diagrama de flujo para el protocolo de ventana corrediza con repetición selectiva.



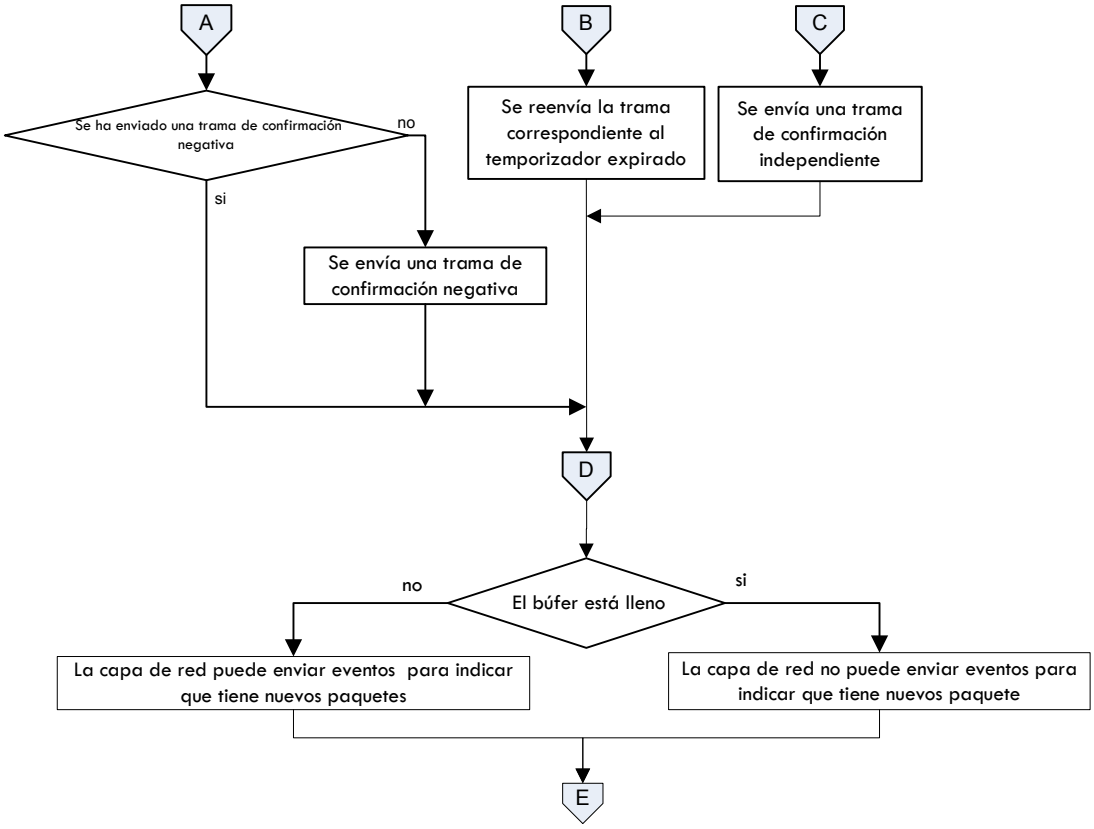


Figura 2.63 Diagrama de flujo para el protocolo de "ventana corrediza de repetición selectiva".

Debido al requerimiento de que haya un temporizador asociado a cada uno de los búferes, se hace uso de arreglo de temporizadores (objeto de la clase *cArray*) de manera similar a lo que se realizó en el protocolo de "ventana corrediza con retroceso a N".

1.5.3.2.1. Módulo

Clase Protocolo6

El objetivo de esta clase es proveer funcionalidad para:

- Proveer funciones que indiquen a la capa de red cuando está lista o no para recibir nuevos paquetes.
- Generar tramas de datos y de control (objetos de la clase *frame*).

- Esperar la llegada de tramas de datos, obtener los paquetes de ellas y almacenarlos en el búfer aún cuando no estén en orden. Una vez que se obtengan los paquetes en orden enviarlos a la capa de red.
- Iniciar un temporizador (copias del puntero *event_timeout*) por cada una de las tramas enviadas y retransmitir sólo las tramas para las cuales expire el temporizador.
- Iniciar un temporizador (puntero *event_acktimeout*) para el caso en que no haya tramas de datos en las que se pueda superponer la confirmación es de recepción y transmitir una trama de control independiente.

En el Código Protocolo 6.1 se presenta la definición de la clase *Protocolo6.*

```
#define MAX_SEQ 7                    // Debe ser 2^n -1 donde n es el
                                     // número de bits
#define NR_BUFS ((MAX_SEQ+1)/2)      // Número de búfers
typedef int seq_nr;
// Enum que será utilizado para asignar el tipo a los eventos que
// se generen
typedef enum{FRAME=1,TIMEOUT=2,READY=3,ACKTIMEOUT=4} tipo_evento;
class Protocolo6: public cSimpleModule
{
private:
// Variables del protocolo
     seq_nr next_frame_to_send;      // Variable que representa
                                     // al número de secuencia de
                                     // la siguiente trama de salida
     seq_nr ack_expected;            // Variable que representa al
                                     // número de confirmación que se
                                     // espera recibir
     seq_nr frame_expected;          // Variable que representa
                                     // al número de secuencia de
                                     // la trama que espera recibir
     frame *r,*s;                    // Variables que representan a las
                                     // tramas que serán recibidas y
                                     // enviadas respectivamente
     packet out_buf[NR_BUFS];        // Arreglo que almacenará los paque-
                                     // tes que van a ser enviados
     packet in_buf[NR_BUFS];         // Arreglo que almacenará los paque-
                                     // tes que van a ser recibido
     seq_nr nbuffered;               // Número de búferes de salida que
                                     // se utilicen actualmente
     int frameSize;                  // Tamaño de la trama
     bool no_nak;                    // Variable que permitirá indicar si
                                     // se ha enviado una trama de tipo
                                     // nak
     seq_nr oldest_frame;            // Variable que representa el número
                                     // de secuencia de la trama cuyo
                                     // temporizador ha expirado
     seq_nr too_far;                 // Límite superior de la ventana
                                     // del receptor
     bool arrived [NR_BUFS];         // Permite indicar si una posición
```

```
                                    // del búfer está llena o vacía
     int i_event;                   // Variable que permitirá verificar
                                    // si el búfer está en posibilidad
                                    // de aceptar nuevos paquetes
     int vtx;                       // Velocidad de transmisión del canal

     int case_;                     // Variable que permite escoger que
                                    // escenario iniciará con la transmi-
                                    // sión, o si iniciarán los dos
                                    // módulos simultáneamente
     double loss;                   // Variable que representa la tasa
                                    // de pérdida de tramas en el canal
     double err;                    // Variable que representa la tasa
                                    // de error de tramas en el canal
     double channel_behavior_err;   // Variable que representa el
                                    // comportamiento del canal a través
                                    // de una distribución de
                                    // probabilidad para el error en las
                                    // tramas
     double channel_behavior_loss; // Variable que representa el
                                    // comportamiento del canal a través
                                    // de una distribución de
                                    // probabilidad para la pérdida de
                                    // tramas
// Variables para los temporizadores
     double timer;                  // Variable que representa el
                                    // intervalo de tiempo que espera
                                    // antes que el temporizador expire
     double TimeEventBefore;        // Almacena el tiempo programado
                                    // para un evento de tipo
                                    // NetworkLayerReady
     double ack_timer_out;          // Variable que representa el inter-
                                    // valo de tiempo que se espera antes
                                    // de enviar una trama de
                                    // confirmación

// Array para los eventos de tipo network_layer_ready
     cArray *arrayNetworkLayerReady;
     cMessage * event_network_layer_ready;
// Array para los eventos de tipo network_layer_ready
     cArray *arrayTimer;
     frame *event_timeout;

// Variable que representa al temporizador de ack
     frame *event_acktimeout;
public:
// Constructor
     Protocolo6();
// Destructor
     virtual ~Protocolo6();
// Funciones del protocolo
     void from_network_layer(packet* buffer);  // Obtiene un paquete de
                                               // la capa de red
     void to_network_layer(packet *msg); // Envía el paquete a
                                         // la capa de red
     void to_physical_layer(frame *s1);  // Envía una trama
                                         // hacia la capa física
     // Función que permite verificar si los valores recibidos
     // caen dentro del rango de valores esperados
     bool between(seq_nr a,seq_nr b,seq_nr c);
```

```
     // Función que realiza el entramado y envía la trama a la capa
     // física
     void send_frame(frame_kind kind,seq_nr frame_nr,
     seq_nr frame_expected,packet buffer[]);


     // Funciones para los temporizadores
     void start_timer(seq_nr aux_frame_nr);    // Inicia el temporizador
                                               // asociado a un búfer

     void stop_timer(int ack_expected);        // Detiene el temporizador
                                               // asociado a un búfer

     void start_ack_timer(); // Inicia un temporizador auxiliar dentro
                             // de cuyo intervalo se puede enviar una
                             // trama de confirmación independiente

     void stop_ack_timer();  // Detiene el temporizador auxiliar

     void enable_network_layer();  // Función que permite indicar
                                   // a la capa de red que puede
                                   // enviar nuevos paquetes

     void disable_network_layer(); // Función que indica a la
                                   // capa de red  que no puede
                                   // recibir más paquetes

// Funciones auxiliares
     void inc(int& k);             // Incrementa circularmente
                                   // el tamaño de la ventana
     void inc_aux(int& k);         // Incrementa circularmente el
                                   // valor del búfer

     void case_network_layer_ready();    // Función que es invocada
                                         // cuando llega un evento
                                         // network_layer_ready

     void case_ack_timeout();// Transmite una trama de confirmación
                             // cuando expira un temporizador de un
                             // ack_timeout

     void case_timeout(cMessage*); // Retransmite una trama
                                   // cuando expira el temporizador

     oid crear_paquete(packet *msg);// Genera un paquete en la capa
                                    // de red

    // Remueve un objeto del array arrayNetworkLayerReady
     void remueve_evento(cMessage *msg);

     // Remueve un objeto del array arrayTimer
     void remueve_evento_timeout(int ev_timeout);


     void registro_evento(); // Imprime el nombre del módulo
                             // y el tiempo en que ocurre un
                             // evento
protected:
// Funciones de la clase cSimpleModule que se redefinen
    virtual void initialize();     // Función que es invocada para
```

```
                                    // iniciar la simulación

    virtual void handleMessage(cMessage *msg);
};
```

Código Protocolo 6.1 Definición de la clase *Protocolo6*.

Variables

- **out_buf[NR_BUFS]:** Arreglo estático de tamaño *NR_BUFS* en el que se almacenarán los paquetes obtenidos de la capa de red que han sido enviados en las tramas correspondientes y que aún no han sido confirmadas.
- **in_buf[NR_BUFS]:** Arreglo estático de tamaño *NR_BUFS* en el que se almacenarán los paquetes que han sido obtenidos de los campos de datos de las tramas recibidas.
- **no_nak:** Variable que representa la bandera que es utilizada para indicar si antes se ha enviado una trama de confirmación negativa.
- **oldest_frame:** Variable que representa el número de secuencia de la trama cuyo temporizador ha expirado.
- **too_far:** Variable que indica el límite superior de la ventana del receptor.
- **arrived[NR_BUFS]:** Arreglo estático con el que se indica si una posición del búfer está llena o vacía.
- **ack_timer_out:** Variable que representa el intervalo de tiempo que se espera para enviar una trama de confirmación independiente.

Funciones

- **send_frame():** Función en la que se realiza el entramado, se verifica si la trama a ser enviada es de datos o de confirmación negativa mediante el campo *kind_var*. Si la trama no es de datos, no se envía información útil, caso contrario se almacena en el campo *info_var* el paquete obtenido de la capa de red, se programa un temporizador y se detiene el temporizador auxiliar. Cabe recalcar que el primer argumento es utilizado para diferenciar el tipo de trama, el valor asignado depende del enumerado *frame_kind* definido para el *message frame* (ver la Sección 2.1.1.2.1).

La implementación de la función *send_frame()* se presenta en el Código Protocolo 6.2.

```
void Protocolo6::send_frame(frame_kind fk,seq_nr frame_nr,seq_nr
frame_expected,packet buffer[])
{    // Se asigna el tipo de trama
     s->setKind(fk);
     // Si la trama es de datos, se coloca un paquete en el
     // campo de datos
     if(fk==data)
     {
           s->setInfo(buffer[frame_nr%NR_BUFS]);
     }
     s->setSeq(frame_nr);
     s->setAck((frame_expected + MAX_SEQ)%(MAX_SEQ+1));
     // Si la trama es de confirmación negativa se configura
     // la bandera no_nak para indicar que ya se envió una
     // trama de este tipo
     if(fk==nak)
     {
           no_nak=false;
     }
     s->setLength(frameSize);
     // Se crea una copia para enviar
     frame *copia=(frame*)s->dup();
     to_physical_layer(copia);
     //Si la trama es de datos se inicia el temporizador
     if(fk==data)
     {
           start_timer(frame_nr);
     }
     // Debido a que se envía una trama que contiene la confirmación
     // de recepción respectiva, se detiene el temporizador
     // de confirmación auxiliar
     stop_ack_timer();
}
```

Código Protocolo 6.2 Implementación de la función *send_frame()* de la clase *Protocolo6*.

- **case_timeout():** Función que será invocada en la función *handleMessage()* cuando se reciba un evento de tipo *TIMEOUT* (valor configurado en la variable *msgkind*). Del temporizador que expira se obtendrá su número de secuencia, ya que éste hace referencia al número de secuencia de la trama al cual pertenece para retransmitirla. Su implementación se presenta en el Código Protocolo 6.3.

```
void  Protocolo6::case_timeout(cMessage *p_r_)
{
           frame * timeout_aux;
     // Convierte el objeto recibido de tipo cMessage a tipo frame
           timeout_aux = check_and_cast<frame*>(p_r_);
```

```
   // Se obtiene el valor de la secuencia de la trama de la
   // cual expiró el temporizador
         oldest_frame=timeout_aux->getSeq();

         ev<<" !!!! Temporizador expirado para la trama
         "<<oldest_frame<<endl;
         bubble("Temporizador expirado");

         // Se remueve el evento del arreglo
         remueve_evento_timeout(oldest_frame%NR_BUFS);
         // Se retransmiten las tramas
         send_frame(data,oldest_frame,frame_expected,out_buf);
}
```

Código Protocolo 6.3 Implementación de la función *case_timeout()* de la clase *Protocolo6*.

- **case_ack_timeout():** Función que será invocada en la función *handleMessage()* cuando se reciba un evento de tipo *ACKTIMEOUT*, en ella *se* enviará la trama de confirmación independiente en el caso que no haya una trama de datos en la cual superponer la confirmación. Su implementación se presenta en el Código Protocolo 6.4.

```
void Protocolo6::case_ack_timeout()
{
     ev<<" Llega el evento AckTimeout"<<endl;
     // Debido a que no es una trama de datos no es necesario asignar
     // un número de secuencia, por esta razón se envía el valor 0
     send_frame(ack,0,frame_expected,out_buf);
}
```

Código Protocolo 6.4 Implementación de la función *case_ack_timeout()*de la clase *Protocolo6.*

- **handleMesssage():** Función que será invocada automáticamente por el *kernel* del simulador cuando llegue un objeto que representa a un evento de tipo: *READY, FRAME, TIMEOUT o ACKTIMEOUT.* El tipo de evento se encuentra determinado por el valor que retorna la función *kind()* de la clase *cMessage.*

  Si el evento recibido es de tipo *READY* (copia del puntero *event_network_layer_ready*) se invoca a la función *case_network_layer_ready*(), se remueve el puntero del arreglo y se lo elimina.

  Si el evento recibido es de tipo *FRAME* se convierte el puntero *p_r_* de tipo *cMessage* a tipo *frame* y se procederá a comprobar si éste está con

error, para lo cual se utilizará la función *hasBitError(). S*i la trama está con error se procede a enviar una trama de confirmación negativa, caso contrario se la acepta y se verifica si corresponde a una trama de datos o de confirmación negativa.

Si la trama es de datos se verifica sus campos de acuerdo a los valores esperados; si el número de secuencia de la trama es diferente a la esperada *(r->getSeq()!=frame_expected && no_nak*) y no se ha enviado una trama de confirmación negativa para ésta, entonces se procede a enviarla. Si el número de secuencia de la trama recibida está en el rango de valores de los números de secuencia de las tramas esperadas, se obtendrá el paquete y se almacenará en el búfer de entrada (*in_buf*), para posteriormente ordenarlos y pasarlos a la capa de red.

Si la trama es de confirmación negativa se procede a enviar lo solicitado en ella, si esta trama se perdiera el funcionamiento del protocolo no se altera.

Posteriormente, se verifica si el valor contenido en el campo de confirmación de la trama está en el rango de valores de confirmación esperados.

Si el evento recibido es de tipo *TIMEOUT*, se invoca a la función *case_timeout()* y se elimina el puntero *p_r_* recibido en el argumento de la función *handleMessage()*.

Si el evento recibido es de tipo *ACKTIMEOUT*, se invoca a la función *case_ack_timeout().*

Finalmente, se verifica si el búfer de salida está lleno, de ser así se invocará a la función *disable_network_layer(),* caso contrario se invocará a la función *enable_network_layer().*

La implementación de la función *handleMessage()* se presenta en el Código Protocolo 6.5.

```
void Protocolo6::handleMessage(cMessage *p_r_)
{
    registro_evento();

    // Se verifica si la trama ha llegado con error
```

```
    // De acuerdo a la variable kind, se identifica que tipo de
    // trama ha llegado
    int event=p_r_->kind();

    switch(event)
    {

/*.................network_layer_ready.................*/
    case READY:
          ev<<" Llega el evento "<<p_r_->name()<<endl;
          case_network_layer_ready();
          remueve_evento(p_r_);
          // Se borra el elemento que llegó
          delete p_r_;
    break;
/*.................frame_arrival.................*/
    case FRAME:
          // Convierte el objeto recibido de tipo cMessage a tipo frame
          r = check_and_cast<frame*>(p_r_);
          // Se verifica si la trama está con error
          if(r->hasBitError()==true)
          {
                ev<<" Llegó una trama con error"<<endl;
                      bubble("Llegó una trama con error");
                if(no_nak)
                {
                      send_frame(nak,0,frame_expected,out_buf);
                }
          }
          else
          {
                ev<<" Recibe la trama (kind,seq,ack) : ("
                <<r->getKind()<<","<<r->getSeq()<<","
                <<r->getAck()<<")"<<endl;
                // Se verifica si es una trama de datos
                if(r->getKind()==data)
                {
// Se hace una comprobación del número de secuencia de la trama
// recibida
// Pueden presentarse dos casos:
// 1.- La trama no es la esperada; si esto ocurre se envía un nak
// (confirmación de recepción negativa) que indica cual es la trama
// que se está esperando, pero para esto se verifica si antes ya se
// ha enviado una nak para dicha trama,ya que no es la intención
// enviar muchas tramas nak
// 2.- La trama es la esperada
                      // Primer caso (La trama no es la esperada)

                      if(r->getSeq()!=frame_expected && no_nak)
                      {
                // Se procede a enviar una trama nak, que indica la
                // trama que no ha  llegado(frame_expected); al no
                // ser una trama de datos no es importante el número
                // de secuencia
                            send_frame(nak,0,frame_expected,out_buf);
                      }
                // Segundo caso (La trama es la esperada)
                // Se inicia un temporizador auxiliar. Si no se ha
                // presentado tráfico de regreso antes de que expire
                // este temporizador, se envía una trama de
```

```
                // confirmación de recepción independiente
                    else start_ack_timer();
                // Sin importar si el número de secuencia de la trama
                // que llega coincide o no con la trama esperada, se
                // realiza la verificación para ver si la trama
                // recibida cae en el rango de los números de secuencia
                // de las tramas que se están esperando;si lo anterior
                // ocurre se verifica si la posición en el búfer en el
                // que se va almacenar la trama no esté ocupada
                    if(between(frame_expected,r->getSeq(),too_far)
                    && arrived[r->getSeq()%NR_BUFS]==false)
                    {
                // Se indica que la posición en el búfer en donde se
                // almacena la trama que ha llegado está ocupada
                            arrived[r->getSeq()%NR_BUFS]=true;
                // Se obtiene el paquete y se lo almacena en el búfer
                // de entrada
                            in_buf[r->getSeq()%NR_BUFS]=r->getInfo();
                            while(arrived[frame_expected%NR_BUFS])
                            {
                        // Se pasa a la capa de red los paquetes, sólo si
                        // están en orden, caso contrario permanecerán
                        // almacenados en el búfer hasta que estén en el
                        // orden esperado
                        to_network_layer(&in_buf[frame_expected%NR_BUFS]);
                             no_nak=true;
                            arrived[frame_expected%NR_BUFS]=false;
                            inc(frame_expected);
                            inc(too_far);
                        // Se inicia un temporizador para enviar una trama
                        // de confirmación independiente en el caso de que
                        // éste expire y aún no se haya enviado la confirmación
                                    start_ack_timer();
                            }
                        }
                    }
                // En el caso de que la trama recibida sea una nak,
                // se verifica si el valor de confirmación contenida
                // en la trama nak cae dentro del rango de valores
                // de las tramas por las que se espera
                // confirmación

                if((r->getKind()==nak) && between(ack_expected,
                (r->getAck()+1)%(MAX_SEQ+1),next_frame_to_send))
                {
                    send_frame(data,(r->getAck()+1)%(MAX_SEQ+1),
                    frame_expected,out_buf);
                }
                // Se verifica si el valor de confirmación contenida
                // en la trama recibida cae dentro del rango de
                // valores de las tramas por las que espera
                // confirmación
                while(between(ack_expected,r->getAck(),next_frame_to_send))
                {
                    nbuffered=nbuffered-1;
                    // Se detiene el temporizador para la trama cuya
                    // confirmación se estaba esperando
                    stop_timer(ack_expected%NR_BUFS);
                    // Se cambia el valor de la trama de quien se
                    // espera confirmación
```

```
                inc(ack_expected);
                ev<<" El valor de ack_expected es:"
                 <<ack_expected<<endl;
          }
     }
          delete p_r_;
     break;
/*.................timeout...................*/
     case TIMEOUT:

          case_timeout(p_r_);
          delete p_r_;

     break;

/*.................acktimeout.................*/
     case ACKTIMEOUT:

          case_ack_timeout();

     break;
     }

     // Se verifica si el búfer está lleno
     if(nbuffered<NR_BUFS){
          // Se habilita para recibir un nuevo paquete de la
          // capa de red
          enable_network_layer();
     }

     else
     {
          // Se niega a recibir nuevos paquetes de la
          // capa de red
          disable_network_layer();
     }

}
```

Código Protocolo 6.5 Implementación de la función *handleMessage()* de la clase *Protocolo6*.

- **start_timer():** Función que permite inicializar el temporizador para una trama dada, en éste caso se hace necesario verificar si dicho temporizador ha sido previamente programado, si éste es el caso se lo cancela y se lo reprograma. La verificación es necesaria debido a que en este protocolo se pueden recibir confirmación de recepción negativa (nak) en solicitud de retransmisión de una trama específica y en este caso se programará un temporizador que aún no ha expirado. En el Código Protocolo 6.6 se presenta la implementación de la función.

```
void Protocolo6::start_timer(seq_nr aux_frame_nr)
{
```

```
     // Número de la trama para la que se activará el temporizador
     int aux_timer=aux_frame_nr%NR_BUFS;
     // Puntero axiliar para programar el temporizador
     frame *p_event_timeout_aux;
     // Nombre del temporizador
     char* timeout_name=(char*)calloc(16,sizeof(char));
     sprintf(timeout_name,"TIMER %d",aux_timer);
     // Se verifica si el temporizador fue
     // programado anteriormente
     if(arrayTimer->find(timeout_name)!=-1)
     {
     int index=arrayTimer->find(timeout_name);
     frame *p_event_timeout_aux = (frame*)arrayTimer->remove(index);
     cancelEvent(p_event_timeout_aux);
     delete p_event_timeout_aux;
     }

     // Se copia el temporizador event_timeout
     p_event_timeout_aux=(frame*)event_timeout->dup();
     // Se asigna un nombre al temporizador
     p_event_timeout_aux->setName(timeout_name);
     // Se asigna el número de secuencia al cual hace
     // referencia el temporizador
     p_event_timeout_aux->setSeq(aux_frame_nr);
     // Se agrega el temporizador al array de temporizadores
     arrayTimer->addAt(aux_timer,p_event_timeout_aux);
     ev<<" Iniciando el temporizador de la trama "
    <<p_event_timeout_aux->getSeq()<<endl;
     // Se programa el temporizador
     scheduleAt(simTime()+timer,p_event_timeout_aux);
     // Se libera espacio de memoria asignado a la
     // variable timeout_name
     free(timeout_name);
}
```

Código Protocolo 6.6 Implementación de la función *start_timer()* de la clase *Protocolo6*.

1.5.3.3. Configuración del escenario de simulación

1.5.3.3.1. Configuración del escenario de simulación en el lenguaje NED

```
simple Protocolo6

    parameters:
        frameSize: numeric,              // Tamaño de la trama
        vtx: numeric,                    // Velocidad de transmisión
        channel_behavior_err: numeric,   // Comportamiento del canal
        channel_behavior_loss: numeric,  // Comportamiento del canal
                                         // para errores
        err: numeric,          // Tasa de errores en las tramas
        loss: numeric,         // Tasa de pérdida de tramas
        case_: string,         // Permite escoger el módulo que empieza
                               // la transmisión

        ack_timer_out: numeric,    // Tiempo en que expirará el
                                   // temporizador de ack
```

```
        timer: numeric;            // Tiempo en que expirará el
                                   // temporizador

    gates:
        out: salida;
        in: entrada;
endsimple
module Protocolo6_

    parameters:
        frameSize: numeric,
        vtx: numeric,
        channel_behavior_err: numeric,
        channel_behavior_loss: numeric,
        err: numeric,
        loss: numeric,
        case_: string,
        ack_timer_out: numeric,
        timer: numeric,
        propagacion: numeric;


   submodules:
        A: Protocolo6;
            parameters:
                frameSize= frameSize,
                timer=timer,
                case_=case_,
                ack_timer_out=ack_timer_out,
                channel_behavior_loss=channel_behavior_loss,
                channel_behavior_err=channel_behavior_err,
                loss=loss,
                err=err,
                vtx=vtx;
            display: "p=35,96;i=device/pc,cyan";
                    B: Protocolo6;
            parameters:
                frameSize= frameSize,
                timer=timer,
                case_=case_,
                ack_timer_out=ack_timer_out,
                channel_behavior_err=channel_behavior_err,
                channel_behavior_loss=channel_behavior_loss,
                err=err,
                loss=loss,
                vtx=vtx;
           display: "p=224,96;i=device/pc,gold";
           connections:
        A.salida --> delay propagacion datarate vtx  --> B.entrada;
        B.salida --> delay propagacion datarate vtx  --> A.entrada;
        display: "b=250,250";
endmodule

network Network : Protocolo6_
endnetwork
```

Escenario Protocolo6 Configuración del escenario para simular el protocolo de “ventana corrediza de repetición selectiva”.

1.5.3.3.2. Configuración de los parámetros en el archivo omnetpp.ini

Adicional al archivo del protocolo anterior, en éste se configura el tiempo para que expire el temporizador de ack.

```
[General]
preload-ned-files=*.ned
network=Network
sim-time-limit = 1s
[Parameters]
Network.frameSize=8000
Network.vtx=1000000          # Velocidad de transmisión en bits por s
Network.timer=0.100          # Tiempo para que el temporizador expire
Network.ack_timer_out=0.010  # Tiempo para que expire el temporizador
                             # de ack
Network.propagacion=0.020    # Tiempo de propagación en el canal en
                             # segundos
Network.loss=0.01            # Tasa de pérdida de tramas
Network.err=0.08             # Tasa tramas con error
Network.case_="A"            #  Módulo  que  empieza  la  transmisión
Network.channel_behavior_err=uniform(0,1)
Network.channel_behavior_loss=uniform(0,1)
[Run 0]
description="Protocolo6"
[Run 1]
description="Protocolo6, empieza A"
Network.case_="A"
[Run 2]
description="Protocolo6, empieza B"
Network.case_="B"
```

Figura 2.64 Configuración de los parámetros en omnetpp.ini para el protocolo de "ventana corrediza de repetición selectiva".

1.5.3.4. Resultados de la simulación

1.5.3.4.1. Impresión en pantalla

A continuación, en la Figura 2.65 se indican algunos de los resultados que se imprimen en la pantalla al realizar la simulación del protocolo "ventana corrediza de repetición selectiva".

```
** Event #0.  T=0.0092000000 (  9ms).  Module #2 `Network.A'
---------------------------------------------
NODO:A, TIME:9.2 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (kind,seq,ack) : (0,0,7)
 Iniciando el temporizador de la trama 0
** Event #1.  T=0.0172000000 ( 17ms).  Module #2 `Network.A'
---------------------------------------------
NODO:A, TIME:17.2 ms
```

```
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Se envía la trama (kind,seq,ack) : (0,1,7)
 Iniciando el temporizador de la trama 1
.
```

```
** Event #4.  T=0.0372000000 ( 37ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:37.2 ms
 Recibe la trama (kind,seq,ack) : (0,0,7)
 Activa el temporizador ack_timeout
 La capa de red recibe:Nuevo Paquete
 Activa el temporizador ack_timeout
** Event #5.  T=0.0452000000 ( 45ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:45.2 ms
 Recibe la trama (kind,seq,ack) : (0,1,7)
 Activa el temporizador ack_timeout
 La capa de red recibe:Nuevo Paquete
 Activa el temporizador ack_timeout
.
.
** Event #11.  T=0.0716000000 ( 71ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:71.6 ms
 Llega el evento NetworkLayerReady
 La capa de red envia un paquete
 Trama con error
 Envía la trama (kind,seq,ack) : (0,3,3)
 Iniciando el temporizador de la trama 3
 Búfer lleno, no se generan eventos
.
.
** Event #18.  T=0.0996000000 ( 99ms).  Module #2 `Network.A'
-------------------------------------------------
NODO:A, TIME:99.6 ms
 Llegó una trama con error
 Envía la trama (kind,seq,ack) : (2,0,2)
.
.
** Event #22.  T=0.1272000000 (127ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:127.2 ms
 Recibe la trama (kind,seq,ack) : (0,6,2)
 Activa el temporizador ack_timeout
** Event #23.  T=0.1275000000 (127ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:127.5 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (kind,seq,ack) : (0,4,3)
 Iniciando el temporizador de la trama 4
** Event #24.  T=0.1352000000 (135ms).  Module #3 `Network.B'
-------------------------------------------------
NODO:B, TIME:135.2 ms
 Recibe la trama (kind,seq,ack) : (2,0,2)
 Envía la trama (kind,seq,ack) : (0,3,3)
 Iniciando el temporizador de la trama 3
.
.
** Event #96.  T=0.4400000000 (440ms).  Module #2 `Network.A'
```

```
-----------------------------------------------
NODO:A, TIME:440 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Pierde la trama
 Iniciando el temporizador de la trama 4
 Búfer lleno, no se generan eventos
.
** Event #120.  T=0.5400000000 (540ms).  Module #2 `Network.A'
-----------------------------------------------
NODO:A, TIME:540 ms
 !!!! Temporizador expirado para la trama 4
 Envía la trama (kind,seq,ack) : (0,4,1)
 Iniciando el temporizador de la trama 4
 Búfer lleno,no se generan eventos
```

Figura 2.65 Impresión en pantalla obtenida de la simulación del protocolo “ventana corrediza de repetición selectiva”.

En los resultados que se presentan en la Figura 2.65 a diferencia del protocolo anterior se puede apreciar:

- Cuando se envía una trama de confirmación negativa en solicitud de retransmisión de una trama que ha llegado con error (por ejemplo al tiempo 99.6ms llega una trama con error y se envia una *nak*).
- Cuando se recibe una *nak* se procede a reenviar la trama solicitada (por ejemplo al tiempo 135.2ms se reenvia la trama solicitada en la nak),

1.5.3.4.2. Visualización gráfica de la simulación

A continuación se presenta los resultados que se visualizan con la simulación del protocolo “ventana corrediza de repetición selectiva”.

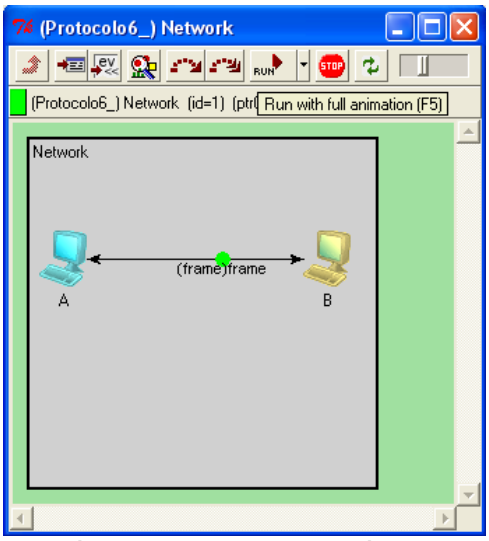


Figura 2.66 Envío de una trama del módulo A al módulo B.

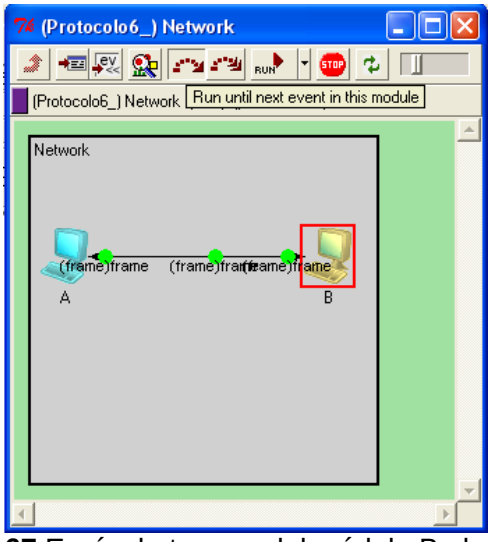


Figura 2.67 Envío de tramas del módulo B al módulo A.

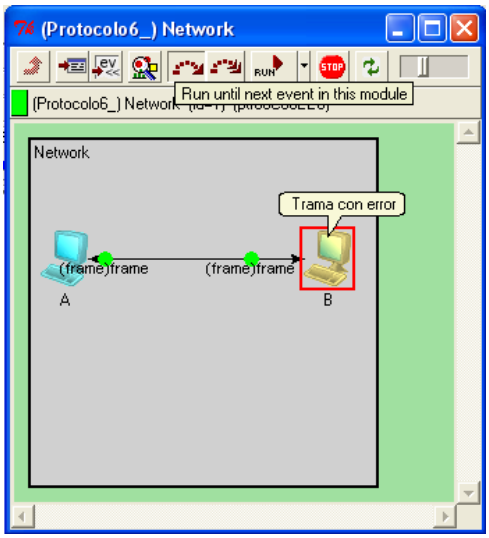


Figura 2.68 La trama a ser enviada es marcada con error.

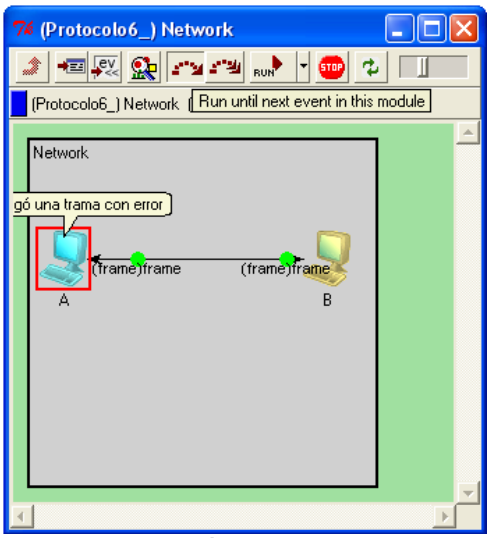


Figura 2.69 Detección de una trama con error.

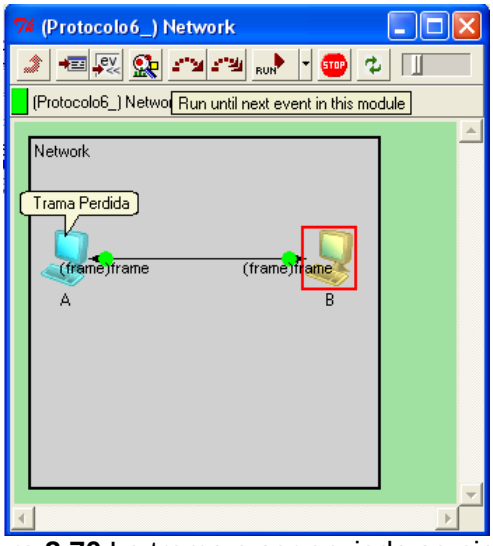


Figura 2.70 La trama a ser enviada se pierde.

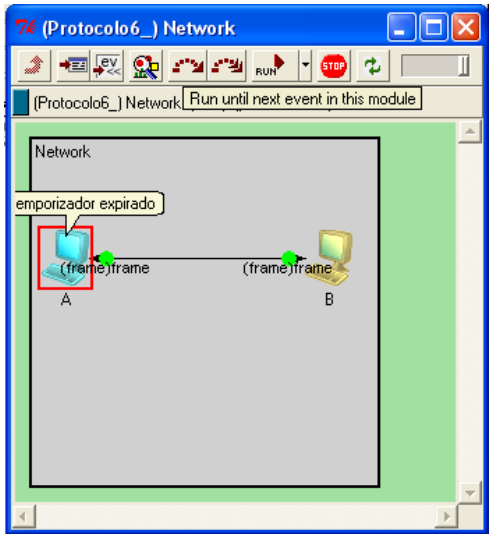


Figura 2.71 Temporizador expirado.

De los resultados obtenidos se puede concluir que el protocolo cumple con las consideraciones planteadas para su implementación, es así que si una trama que llega es detectada con error ésta es descartada e inmediatamente se informa a través de una trama de control que se la retransmita; si la trama se pierde en el canal, el temporizador expira y se procederá a la retransmisión; ya no hay la dependencia de que haya datos en la que se pueda superponer el acuse de recibo, por tanto esta confirmación será independiente y, finalmente cumple con la naturaleza propia del protocolo que es la de retransmisión selectiva.

Con el cumplimiento de esas condiciones se puede garantizar que es el protocolo que mejor ocupa el canal de comunicación.

1.6. INTERACCIÓN DEL SIMULADOR CON NUEVOS PROTOCOLOS IMPLEMENTADOS

Antes de describir la interacción del simulador con los nuevos protocolos implementados se describen los componentes del simulador y su interacción **[1]**.

Componentes del simulador

- ***Sim*:** *Kernel* del simulador, encargado de la planificación de eventos futuros.
- ***Envir*:** Interfaz común para las interfaces de usuario *Tkenv* y *Cmdenv*.
- ***Cmdenv* y *Tkenv*:** Interfaces de usuario específicas en las que se presentarán los resultados obtenidos de la simulación.
- **Componentes del modelo:** Clases que representan a cada uno de los elementos que forman parte del **modelo de ejecución**[1].

Interacción entre los componentes

En la Figura 2.72 se presentan los componentes del simulador y su interacción.

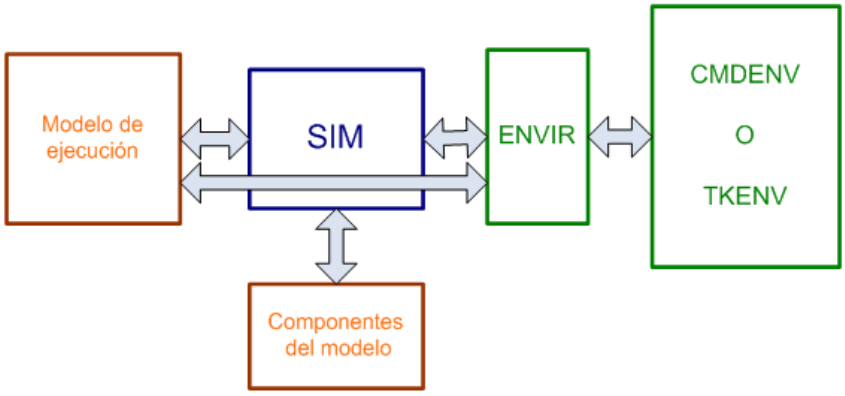


Figura 2.72 Componentes del simulador OMNET++**[1]**.

- ***Sim* vs componentes del modelo:** El *kernel* del simulador crea el modelo de ejecución, lo que consiste en instanciar objetos de las clases que conforman los componentes del modelo.

[1] También llamado como escenario de simulación.

- **Modelo de ejecución vs *Sim*:** El *kernel* del simulador maneja la planificación de eventos para los objetos que conforman el modelo de ejecución.
- **Modelo de ejecución vs *Envir*:** *Envir* se presenta como una interfaz entre el modelo de ejecución y la interfaz de usuario (*Tkenv* ó *Cmdenv*) a través del objeto *ev*[1] para presentar mensajes en la interfaz.
- ***Sim* vs *Envir*:** *Envir* indica al *kernel* del simulador el modelo de ejecución que debe crear.
- ***Envir vs Tkenv* y *Cmdenv*:** Desde la función *main()* (parte de *Envir*), con el objeto *ev* se invoca a la interfaz de usuario apropiada sea *Tkenv* o *Cmdenv* .

1.6.1. CÓDIGO GENERADO POR EL COMPILADOR NEDTOOL

Como se explicó en la Sección 1.2.2, los módulos que forman parte del modelo de ejecución se encuentran definidos en lenguaje NED y su implementación en lenguaje C++, es así que cada módulo simple que se defina en lenguaje NED tendrá su correspondiente clase que implementa su funcionalidad.

En cuanto a los módulos compuestos definidos en lenguaje NED, éstos no son implementados en C++ por el usuario, ya que su comportamiento depende de la implementación de los módulos simples que lo componen; sin embargo, tienen su respectiva clase en C++ que es generada por el compilador *nedtool*.

Con el compilador *nedtool* de OMNET++, a partir de un archivo .ned se genera un archivo .cpp que contiene la definición e implementación de las clases correspondientes a los módulos compuestos y *networks,* definidos en lenguaje NED.

Para la explicación se ha tomado como ejemplo el archivo protocolo1.ned en el que se ha definido el modelo para el protocolo "*simplex* sin restricciones", una vez compilado este archivo, las principales partes del archivo protocolo1_n.cpp generado son:

---

[1] Objeto global de la clase *cEnvir.*

- La clase correspondiente para el módulo compuesto *Protocolo1* (nombre asignado al módulo compuesto en lenguaje NED), la cual es derivada de la clase *cCompoundModule*; la única función que se redefine en esta clase es *doBuildInside()*, la cual es invocada en la función *setupNetwork()* de la clase *Network* para construir los submódulos y las conexiones interiores al mismo (ver Figura 2.73).

```
class Protocolo1 : public cCompoundModule
{
  public:
    Protocolo1() : cCompoundModule() {}
  protected:
    virtual void doBuildInside();
};
```

```
// Lenguaje NED
module Protocolo1
      parameters:
        . . .
      submodules:
        . . .
      connections:
        . . .
endmodule
```

Figura 2.73 Definición del clase *Protocolo1* que representa al módulo compuesto.

- La clase derivada de *cNetworkType* de nombre *Network* (nombre asignado a la red en lenguaje NED), en la cual se implementa la función *setupNetwork()* que se encargará de construir la red *Network* (ver Figura 2.74).

```
// Lenguaje NED
// Protocolo1 nombre del módulo
// compuesto previamente definido
network Network: Protocolo1
endnetwork
```

```
class Network : public cNetworkType
{
  public:
// Envía como argumento el nombre de la red que se le pasa desde
// omnetpp.ini
    Network(const char *name) : cNetworkType(name) {}

    Network(const Network& n) : cNetworkType(n.name()) {operator=(n);}
     // Función que es invocada para crear la red que está formada por
     // el módulo compuesto Protocolo1
    virtual void setupNetwork();
};
```

Figura 2.74 Definición del clase *Network* que representa al módulo de la red.

- Llamada a los macros para las clases antes mencionadas (ver Figura 2.75) y que se explicará en la Sección 2.3.2.

```
Define_Module(Protocolo1);  // Protocolo1: clase que representa al
                            // módulo compuesto
Define_Network(Network);    // Network: clase que representa a la red
```

Figura 2.75 Llamada a los macros para las clases *Protocolo1* y *Network*.

- Funciones globales que serán utilizadas en la creación de los módulos y cuya funcionalidad será explicada en la Sección 2.3.3. Las principales funciones son: *_getModuleType(), _readModuleParameters(), _checkGate().*

Una vez que se tienen las clases correspondientes en C++ (que conforman los componentes del modelo) para el modelo de ejecución, éstas podrán interactuar con los componentes del simulador y de esta manera llevar a cabo la simulación.

1.6.2. DESCRIPCIÓN DEL PROCESO DE INTERACCIÓN DEL SIMULADOR CON UN NUEVO PROTOCOLO

En tiempo de compilación se sustituyen los macros para las clases que se especifican en su argumento por líneas de código previamente definidas. En la Figura 2.76 se indica el código por el cual fue sustituido el macro Define_Module(Protocolo1), y las funciones que el macro representa serán invocadas por la función *executeAll()* que será explicada posteriormente.

```
/* Código que sustituirá al macro Define_Module(Protocolo1) */
static cModule *Protocolo1__create(const char *name, cModule
*parentmod)
{    return new Protocolo1(name, parentmod);
}
// Macro que será sustituido por el código que será invocado para la
// creación de la lista global modtype
EXECUTE_ON_STARTUP(Protocolo1__mod, modtypes.instance()->add(new
cModuleType("Protocolo1 "," Protocolo1",(ModuleCreateFunc)
Protocolo1__create));
)
```

Figura 2.76 Código que sustituirá a los respectivos macros.

Desde la función *main()* que forma parte de *Envir*, se invoca la función *setup()* con el objeto global *ev* de la clase *cEnvir,* como se muestra en la Figura 2.77.

```
ENVIR_API int main(int argc, char *argv[])
{
      cStaticFlag dummy;

      printf("OMNeT++/OMNEST Discrete Event Simulation  (C)
      1992-2005  Andras Varga\n");
      printf("Release:"OMNETPP_RELEASE", edition:"OMNETPP_EDITION".\n");
      printf("See the license for distribution terms and warranty
      disclaimer\n");
      // Invoca a la función setup() de cEnvir
      ev.setup(argc,argv);
      int ret = ev.run();
      ev.shutdown();
      printf("\nEnd run of OMNeT++\n");
      return ret;

}
```

Figura 2.77 Función *main()* del simulador OMNET++.

Dentro de la función *setup()* se invoca a la función estática *executeAll()* de la clase *ExecuteOnStartup,* la cual invocará a las funciones creadas a partir del macro *EXECUTE_ON_STARTUP* con lo cual se creará un nuevo objeto de tipo *cModuleType*[1] para un módulo dado, en este caso *Protocolo1*, a su vez el objeto creado es añadido a la lista global *modtypes*[2] (lista de objetos de tipo *cModuleType*). Este proceso se realiza para cada uno de los módulos que conforman el modelo de ejecución.

De manera similar, se realiza el proceso antes descrito para la clase *Network*; en este caso se añadirá un objeto de nombre *Network* a otra lista global: *networks* (lista de objetos de tipo *cNetworkTypes*).

Posteriormente, se lee el contenido del archivo omnetpp.ini mediante la función *readFile()* de la clase *cIniFile* para determinar la interfaz donde se presentarán los resultados de la simulación que en este caso será *Tkenv*.

Una vez escogida la interfaz *Tkenv* (representada por la clase *TOmnetTkApp*) se invoca a su respectiva función *setup()* cuyo objetivo es configurar dicha interfaz; además, en ella se invoca a la función global *createTkCommands()* cuya finalidad es asociar los procedimientos Tcl con funciones en C++ de acuerdo al arreglo *tcl_commands[]* que se presenta en la Figura 2.78.

---

[1] Un objeto de tipo cModuleType actuará como una fábrica, cuando la función create() es invocada, éste producirá el nuevo módulo.

[2] Objeto de tipo *cSingleton* que internamente contiene un objeto *cArray* en el cual almacerá los objetos de tipo *cModuleType.*

```
OmnetTclCommand tcl_commands[] = {
   // Commands invoked from the menu
   { "opp_newnetwork",       newNetwork_cmd     }, // args: <netname>
   { "opp_newrun",           newRun_cmd         }, // args: <run#>
   { "opp_createsnapshot",   createSnapshot_cmd }, // args: <label>
   { "opp_exitomnetpp",      exitOmnetpp_cmd    }, // args: -
   { "opp_onestep",          oneStep_cmd        }, // args: -

   { "opp_run",         run_cmd            }, // args: ?fast?
                                              // ?timelimit?
   { "opp_onestepinmodule",  oneStepInModule_cmd}, // args:
                                                   // <inspectorwindow>
   { "opp_set_run_mode",     setRunMode_cmd     }, // args:
                                                   // fast|normal
                                                   // |slow|express
   { "opp_set_run_until",    setRunUntil_cmd    }, // args: <timelimit>
                                                   // <eventlimit>
   { "opp_set_run_until_module",setRunUntilModule_cmd}, // args:
                                                    // <inspectorwindow>
   { "opp_rebuild",          rebuild_cmd        }, // args: -
   { "opp_start_all",        startAll_cmd       }, // args: -
   { "opp_finish_simulation",finishSimulation_cmd}, // args: -
   { "opp_loadlib",          loadLib_cmd        }, // args: <fname>
                                                   // Utility commands
   { "opp_getrunnumber",     getRunNumber_cmd   }, // args: -  ret:
                                                   // current run
.
.
.
   { "opp_loadnedfile",         loadNEDFile_cmd }, // args: <ptr> ret:
                                                 // <xml> experimental
   { "opp_colorizeimage",       colorizeImage_cmd      },   // args:
                                                    // <image> ... ret:
-
                                                    // end of list
   { NULL, },
};
```

Figura 2.78 Parte del arreglo *tcl_commands[].*

Continuando con la ejecución de la función *main()*, en ella se invocará la función *run()* de la clase *cEnvir*, que a su vez invocará a la función *run()* de la clase *TOmnetTkApp*, dentro de esta función se invocará al procedimiento Tcl *startup_commands{}*[1]. Dentro del procedimiento *startup_commands{}* se invoca el comando *opp_newrun* al cual está asociado la función global *newRun_cmd()*.

La función *newRun_cmd()* invoca a la función *newRun()* de la clase *TOmnetTkApp* (ver la Figura 2.79) dentro de la cual se invoca a la función *readPerRunOptions()* de la clase base TOmnetApp, la misma que provee funcionalidad para obtener los valores de los parámetros generales para el

[1] Procedimiento implementado en el archivo main.tcl

modelo de ejecución, como por ejemplo se obtiene el nombre de la red, el tiempo de inicio, el tiempo de finalización de la ejecución de la simulación y otros,. ver la Figura 2.80.

```
void TOmnetTkApp::newRun(int run)
{
    try
    {
        // Configura una nueva red
        run_nr = run;
        readPerRunOptions(run_nr);
/*
La función opt_network_name.c_str()devuelve el nombre de la red que se
desea configurar; este nombre se envía como argumento de la función
findNetwork() la cual buscará en la lista global networks el objeto
encargado de la creación de la red respectiva (Network) y devolverá un
puntero a éste.
*/
        cNetworkType *network = findNetwork(opt_network_name.c_str());
        if (!network)
        {
            CHK(Tcl_VarEval(interp,"messagebox {Confirm} {Network '",
opt_network_name.c_str(), "' not found.} info ok",NULL));
            return;
        }
        CHK(Tcl_VarEval(interp, "clear_windows", NULL));
// Se invoca a la función setupNetwork de la clase cSimulation enviando
// como argumento el objeto de tipo cNetworkType
        simulation.setupNetwork(network, run_nr);
        startRun();
        simstate = SIM_NEW;
    }
    catch (cException *e)
    {
    }
     animating = false; // affects how network graphics is drawn!
    updateNetworkRunDisplay();
    updateNextModuleDisplay();
    updateSimtimeDisplay();
    updateInspectors();
}
```

Figura 2.79 Función *newRun()* de la clase *TOmnetTkApp.*

```
void TOmnetApp::readPerRunOptions(int run_no)
{
    cConfiguration *cfg = getConfig();
    const char *section = getRunSectionName(run_no);

    // Se obtiene el nombre de la red, si no existe, se da el nombre por
    // defecto “default”
    opt_network_name = cfg->getAsString2(section, "General", "network",
    "default");
    opt_simtimelimit = cfg->getAsTime2(section, "General", "sim-time-
    limit", 0.0);
. . . .
}
```

Figura 2.80 Parte de la función *readPerRunOptions()* de la clase *TOmnetApp*.

Posteriormente, con el objeto global *simulation* de la clase *cSimulation* se invoca a la función *setupNetwork()* enviando como argumento el objeto de tipo *cNetworkType* encargado de la creación de la red *Network* como se presenta en la Figura 2.80.

Dentro de la función *setupNetwork()* de la clase *cSimulation* se invoca a la función *setupNetwork()* de la clase *Network* (clase derivada de *cNetworkType*) que se presenta en la Figura 2.81.

```
static cModuleType *_getModuleType(const char *modname)
{
    cModuleType *modtype = findModuleType(modname);
    return modtype;
}

void Network::setupNetwork()
{
   // Variables temporales:
   cPar tmpval;
   const char *modtypename;

   cModuleType *modtype;

   modtype = _getModuleType("Protocolo1");
   cModule *Network_p = modtype->create("Network", NULL);
     .
     .
     .
   Network_p->buildInside();
}
```

Figura 2.81 Función *setupNetwork()* de la clase *Network*.

La función *setupNetwork()* invocará a la funcion *_getModuleType()*, la cual de acuerdo a su argumento (*Protocolo1*), buscará el objeto de tipo *cModuleType* encargado de la creación del módulo respectivo (*Protocolo1*) y devolverá un puntero a éste.

Con el objeto estático de tipo *cModuleType* obtenido de la lista global correspondiente (*modtypes*), se invocará a la función *create(),* la cual configura el módulo respectivo con el nombre *Network* y además asigna un identificador único para el módulo.

Posteriormente, se invocará a la función *buildInside()* para el módulo compuesto (*Protocolo1*). La función *buildInside()* a su vez invocará a la función

doBuildInside() (ver la Figura 2.82) que es la encargada de construir los submódulos y las conexiones internas que conforman el módulo compuesto **[1]**.

```
void Protocolo1::doBuildInside()
{
      . . . .
         // Submódulos:
    cModuleType *modtype = NULL;
     /*************** Submódulo 'A'******************/
// Se busca un objeto de tipo cModuleType de nombre Protocolo1Sender
    modtype = _getModuleType("Protocolo1Sender");
    cModule *A_p = modtype->create("A", mod);
    {
// Se asignan los parámetros al submódulo de nombre A
        A_p->par("tamaniobuffer") = mod->par("tamaniobuffer");
        A_p->par("frameSize") = mod->par("frameSize");
     }
    /*************** Submódulo 'B'******************/
// Se busca un objeto de tipo cModuleType de nombre Protocolo1Receiver
    modtype = _getModuleType("Protocolo1Receiver");
    cModule *B_p = modtype->create("B", mod);
    {

    }
    /************** Conexiones**********************/
// Se configuran las compuertas de entrada y salida a los módulos
    srcgate = _checkGate(A_p, "salida");
    destgate = _checkGate(B_p, "entrada");
    // Se crea el canal de comunicación
    channel = new cBasicChannel("channel");
// Se configuran los parámetros del canal de comunicación antes
// creado
    par = new cPar("delay");
    (*par) = mod->par("propagacion")    ;
    channel->addPar(par);
    par = new cPar("datarate");
    (*par) = mod->par("vtx")    ;
    channel->addPar(par);
// Se conectan las compuertas de los módulos
    srcgate->connectTo(destgate,channel);
// Se construyen los submódulos A y B que forman parte parte del
// módulo compuesto Protocolo1
    A_p->buildInside();
    B_p->buildInside();
}
```

Figura 2.82 Parte de la función *doBuildInside()* de la clase *Protocolo1*.

Continuando con la función *newRun()* de la clase *TOmnetTkApp* presentada en la Figura 2.79 se invoca a la función *startRun()* de la misma clase, que a su vez invoca a la función *startRun()* de la clase *cSimulation* donde se invoca a la función *callInitialize()* de la clase *cModule* y ésta a su vez invoca a la funcion *initialize()* para cada uno de los módulos del modelo de ejecución. En la función *initialize()* se puede especificar o no el evento que el *kernel* del simulador debe programar.

Una vez que ocurra el evento planificado, el *kernel* del simulador invoca a la función *handleMessage()* del módulo que debe procesarlo.

Finalmente, se ejecuta la función *shutdown()* de la clase cEnvir invocada en la función *main()* (ver la Figura 2.77), la cual es invocada antes de salir del programa de simulación, ésta a su vez invoca a la función *shutdown()* de la clase *cSimulation* donde se elimina el modelo de ejecución y se eliminan los eventos futuros programados para el modelo **[1]**.

1.6.3. DIAGRAMA DE SECUENCIA

En la Figura 2.83 se presenta el diagrama de secuencia de la interacción del simulador con el nuevo protocolo y que resumen el proceso que se describe en la sección anterior.

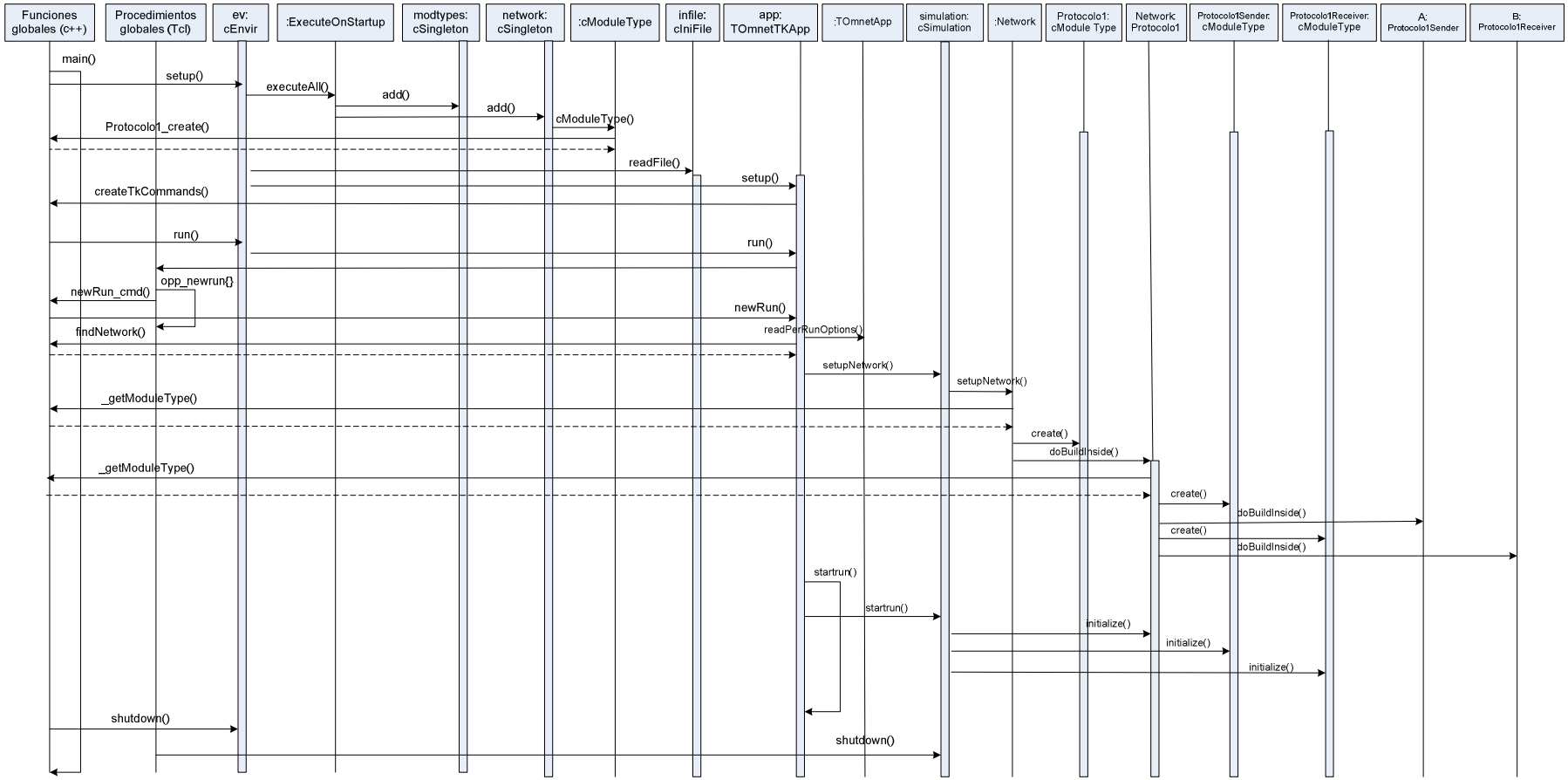


Figura 2.83 Diagrama de secuencia de la interacción del simulador con el nuevo protocolo.

CAPÍTULO 3

IMPLEMENTACIÓN EN NS-2

Todos los elementos (protocolos, dispositivos, conexiones, etc.) que intervienen en un escenario de simulación de red son considerados como objetos de red (ver el Anexo D).

IMPLEMENTACIÓN DE UN PROTOCOLO EN NS-2

Para llevar a cabo la implementación de un protocolo en NS-2 es necesario tener una visión clara de las clases de las que depende su funcionamiento.

La Figura 3.1 presenta parte de la jerarquía de clases compilada de NS-2 de la que formará parte la nueva clase que representará al protocolo a implementarse.

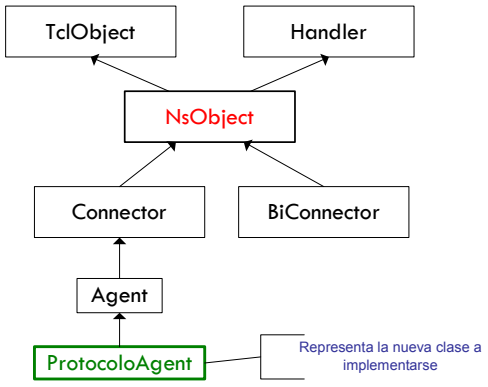


Figura 3.1 Parte de la Jerarquía Compilada.

Clase NsObject

Es una clase abstracta, derivada de las clases *TclObject* y *Handler*; las principales funciones que son heredadas y redefinidas son *command()* y *handle()*, respectivamente. A partir de *NsObject* se derivan las clases que representan a los objetos que forman parte de la ruta de datos[1].

Las principales funciones que provee *NsObject* son:

- **handle():** Función redefinida de la clase *Handler* que permite la manipulación de eventos.
- **recv():** Función virtual pura que será redefinida por las clases que se deriven de ella.
- **command():** Función redefinida de la clase *TclObject*.

Clase Connector

Connector es la clase base para implementar objetos de red sencillos que tienen una sola salida, como son: colas, retardos, agentes; con la combinación de algunos de ellos se pueden formar objetos de red básicos, como por ejemplo nodos y enlaces.

Esta clase tiene un puntero de nombre *target_* que representa al objeto de red al cual está conectado (también derivado de clase *NsObject*), con este puntero se invocará a la función *recv()* de la clase respectiva.

Las principales funciones que esta provee son:

- **send():** Función que permite enviar el flujo de datos en una sola dirección.
- **recv():** Función redefinida de la clase *NsObject* cuyo principal objetivo es recibir el evento generado hacia él.

Clase BiConnector

BiConnector es la clase base para implementar objetos de red sencillos que tienen dos salidas, como son: LL, IFq, MAC, etc; como se observa en la Figura

[1] Se construye mediante la interconexión de los componentes de red, indicando en cada objeto cuál es el siguiente objeto de destino.

3.2. Éstos forman parte de un nodo móvil utilizado para simular protocolos de redes inalámbricas.

Esta clase tiene dos punteros de tipo *NsObject* llamados *uptarget_* y *downtarget_* que son utilizados por las funciones *sendUp()* y *sendDown()* (funciones propias de esta clase), a través de los cuales se envía el flujo de datos ya sea en una u otra dirección **[2]**.

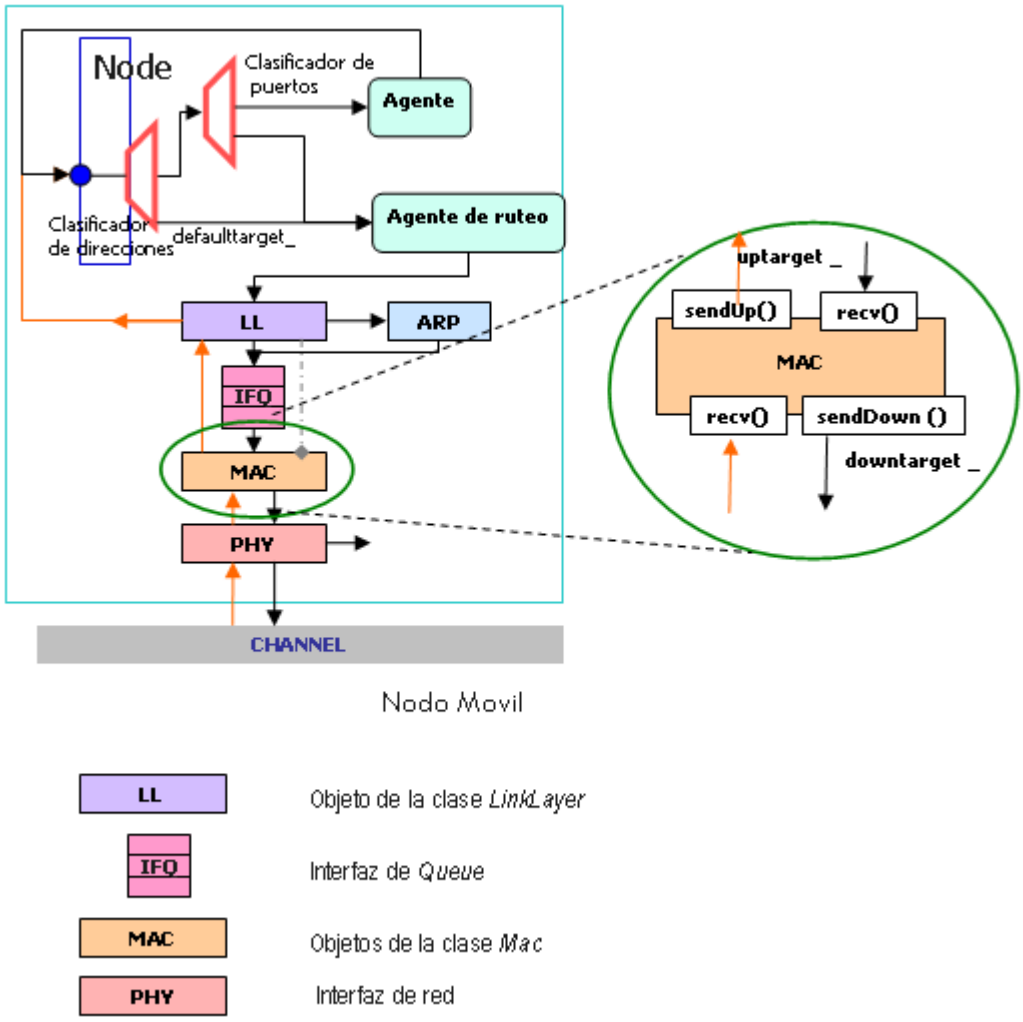


Figura 3.2 Esquema de un nodo móvil **[10]**.

Por la funcionalidad que esta clase provee, no es utilizada en este proyecto, pero se la menciona para indicar la posibilidad de desarrollar protocolos para medios inalámbricos.

Clase Agent

La finalidad de esta clase es proveer funciones que permitan manipular las unidades de datos[1] (su explicación se la realizará en la Sección 3.1.1), es decir, crearlas y/o destruirlas.

A través de la implementación de una clase derivada de *Agent* se puede especificar el comportamiento de un protocolo ante la llegada de información. La principal función que se define en esta clase es *send().* Adicionalmente, se define la función *allocpkt()* que será explicada en la Sección 3.1.1.1.2.

En la Figura 3.3 se presenta la estructura de un agente en NS-2.

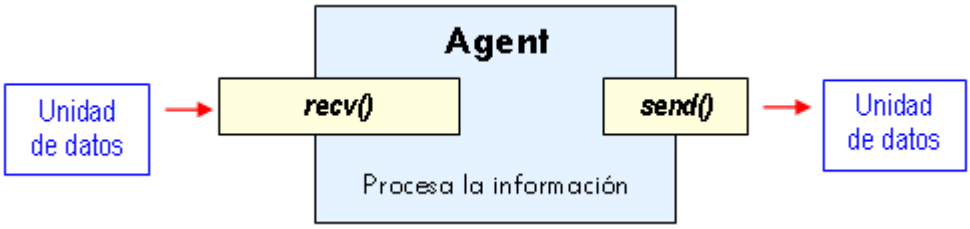


Figura 3.3 Estructura de un agente.

1.7. IMPLEMENTACIÓN DE PROTOCOLOS ELEMENTALES DE ENLACE DE DATOS

Los protocolos que se implementan a continuación se presentan de forma secuencial de acuerdo a un grado de complejidad creciente, lo que requiere el uso de nuevas funciones y variables; por tanto, se ha considerado que en cada protocolo se explicará únicamente lo que se agregue a la implementación respecto al protocolo anterior. Además, la implementación de los protocolos se realizará de acuerdo a los algoritmos correspondientes indicados en los diagramas de flujo que se presentan en el Capítulo 2.

[1] Estructuras de datos a las cuales se accede a través de objetos de tipo *Packet*.

1.7.1. PROTOCOLO SIMPLEX SIN RESTRICCIONES

1.7.1.1. Diseño e implementación en C++

Para implementar este protocolo es necesario definir una unidad de datos y dos agentes que la procesen.

1.7.1.1.1. Unidad de datos

En NS-2, la unidad fundamental para el intercambio de información es un objeto de la clase *Packet*[1] (clase derivada de la clase *Event*), el mismo que está compuesto de un conjunto de cabeceras y una sección de datos que usualmente no es utilizada, ya que el intercambio de datos reales no es el centro de interés en las simulaciones (ver la Figura 3.4).

Cada cabecera representa una unidad de datos de determinado protocolo, es así que por ejemplo la cabecer *ip header*, representa al paquete *ip* el cual es utilizado como unidad de intercambio de información en el protocolo *ip*.

La implementación de cada una de las cabeceras se la realiza mediante una estructura de datos.

Clase Packet

En la Figura 3.4 se presenta el esquema de un objeto de la clase *Packet*, en donde se indican las principales variables que son:

- **next_:** Puntero a un objeto de la clase *Packet*, el cual es utilizado para crear una lista enlazada de objetos de tipo *Packet*.
- **hdrlen_:** Variable que representa el tamaño en memoria que se reserva para las cabeceras de los protocolos, este tamaño es definido durante la inicialización del simulador.
- **bits_**: Puntero al primer *byte* de la zona de memoria reservada para las cabeceras de todos los protocolos implementados en el simulador (aquellos que ya vienen por defecto implementados y los nuevos que agrega el programador).

[1] Clase propia del simulador NS-2

- **data_**: Variable que representa la zona de memoria para los posibles datos propiamente dichos**.**

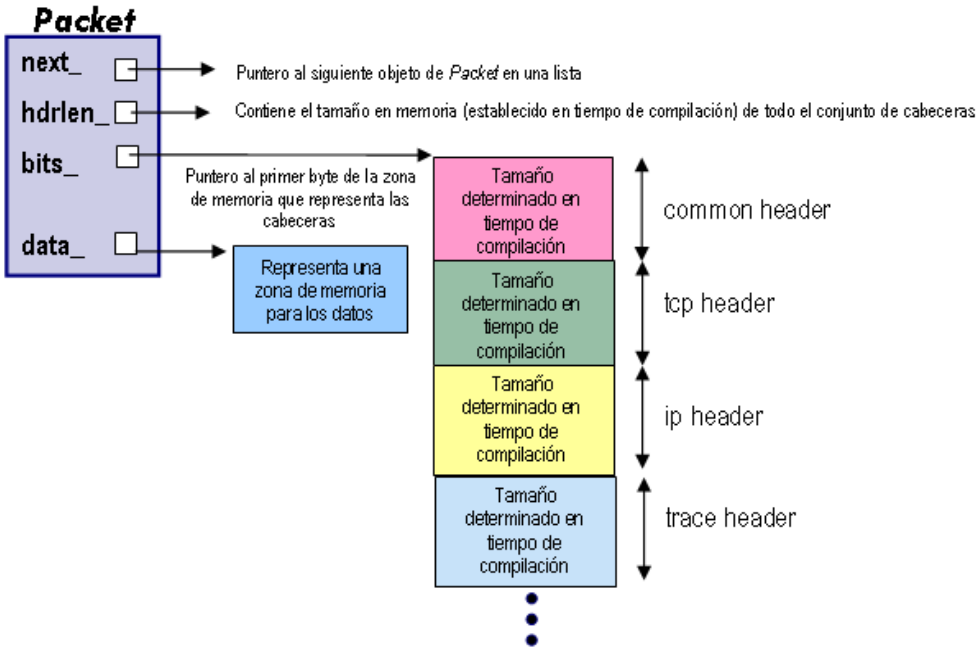


Figura 3.4 Objeto de la clase *Packet* **[6]**.

Cabe mencionar que existe la implementación de una cabecera común (*header common*), que es representada a través de la estructura *hdr_cmn* y es utilizada por todos los protocolos aún cuando se hayan definido cabeceras propias para cada uno de ellos.

A continuación se describen las principales funciones que esta clase ofrece y que son utilizadas para la implementación de los protocolos.

- **alloc():** Función utilizada para la asignación de espacio de memoria cuando se requiera crear nuevos objetos de la clase *Packet*. Esta función es invocada por la función *allocpkt()* implementada en la clase *Agent*.
- **free():** Función utilizada para liberar el espacio de memoria reservada para la variable *data_* de esta clase.
- **access():** Función que retorna el contenido del arreglo *bits_* en una posición dada.

Estructura hdr_cmn

Estructura propia del simulador NS-2. Las variables miembro definidas en esta estructura se utilizan para el análisis del flujo de paquetes; a continuación se describen las principales variables **[2]**:

- **size_:** Variable utilizada para modelar el tamaño en *bytes* de la unidad de datos*;* no tiene relación con la reserva de memoria que hace el compilador para la estructura.
- **ts_:** Variable utilizada para representar el retardo de encolamiento en el clasificador del nodo.
- **ptype_:** Variable de tipo *packet_t* (tipo enumerado definido para la clase *p_info,* que será explicada posteriormente), utilizada para identificar el tipo de paquete.
- **uid_:** Variable que representa un identificador único, es utilizada por el planificador de eventos para fijar la llegada de las unidades de datos.
- **error_:** Variable utilizada como bandera de error.

Las variables de la estructura *hdr_cmn* son utilizadas en función de los requerimientos del protocolo.

Estructura hdr_frame

Para los nuevos protocolos a desarrollarse se ha definido el formato de una unidad de datos representado mediante la estructura *hdr_frame* la misma que contiene los campos mostrados en la Figura 3.5 y, a la cual se accederá a través de un objeto de la clase *Packet*.

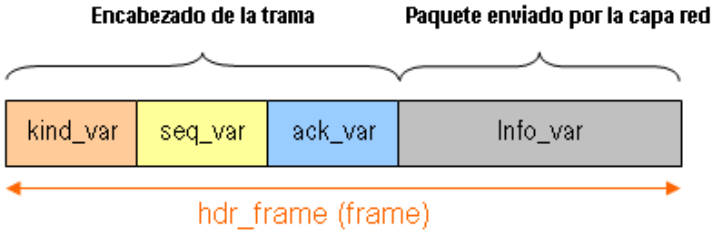


Figura 3.5 Campos de la estructura *hdr_frame*.

La definición de la estructura *hdr_frame* se presenta en el Código 3.1. Para la estructura *hdr_frame* se ha definido el nombre *frame* con el que se hará referencia a la trama en posteriores explicaciones.

```
/* Definición del tipo de dato con el cual se identificará el
 * tipo de trama */
      enum frame_kind
     { data,      // Si la trama es de datos
       ack        // Si la trama es de control
     };
/* Estructura que representa a la trama*/
typedef struct hdr_frame{
/* Variables que representan los campos de la trama*/
       frame_kind kind_var;    // Tipo de trama
       int seq_var;            // Número de secuencia
       int ack_var;            // Número de confirmación
       paquete info_var;       // Campos de datos, el tipo paquete se
                               // define en Código Paquete
/* Variable y función que se utilizará para acceder a la trama
 * dentro de un objeto de tipo Packet*/
      static int offset_; // Variable que permite acceder a la trama
/* Función que permite acceder a la trama */
      inline static hdr_frame* access(Packet* p) {
            return (hdr_frame*) p->access(offset_); }
}frame;// Nombre con el que se identifica a la estructura hrd_frame
```

Código 3.1 Definición de la estructura *hdr_frame*.

Los campos definidos para la unidad de datos son: número de secuencia (*seq_var*), número de confirmación (*ack_var*), tipo de trama (*kind_var*) y campo de datos (*info_var*). Adicionalmente, se define una variable estática cuyo nombre es *offset_* que apunta al primer *byte* de memoria reservada para la estructura *hdr_frame* y que será utilizada para acceder a ella; también se ha definido la función *access(),* la misma que retorna un puntero a *hdr_frame* con el que se podrá acceder a sus campos para realizar modificaciones.

El campo de datos de *hdr_frame* es representado mediante una estructura de nombre *paquete*[1] que representa a la unidad de datos que es entregada por la capa de red, o que se envía a la capa de red. En el Código 3.2 se presenta la estructura *paquete*.

```
struct paquete{   char * data;
};
```

Código 3.2 Definición de la estructura *paquete*.

[1] No hace referencia a la clase *Packet* u objetos de ella.

Clase p_info

Es una clase propia del simulador NS-2, que permite identificar una unidad de datos con su nombre simbólico.

La nueva unidad de datos (estructura) que se cree será identificada con un código numérico, el mismo que deberá ser agregado en la enumeración *packet_t* **[11]**. Por otra parte, en el constructor de la clase *p_info* se debe agregar el nombre simbólico de acuerdo a su respectivo código numérico, como se presenta en el Código 3.3, donde *PT_FRAME* representa el código numérico y *frame* representa el nombre simbólico.

```
/* Enumerado packet_t */
enum packet_t {

PT_TCP,
...
...
...
...
...

...
PT_FRAME        // Código numérico que ha sido agregado para
                // representar a la cabecera hdr_frame
};

/* Parte de la clase p_info*/
class p_info {

public:

p_info() {
name_[PT_TCP]= "tcp";
...
name_[PT_FRAME]= "frame";   // Asignación del nombre simbólico frame
                            // al código numérico PT_FRAME que será
                            // utilizado para identificar a la
                            // estructura hdr_frame

}
}
```

Código 3.3 Parte de la implementación de la clase *p_info*.

Clase FrameHeaderClass

Clase definida para permitir la vinculación entre los espacio C++ y OTcl de la estructura *hdr_frame*, cuyo proceso será explicado posteriormente en la Sección

3.2.2. En el Código 3.4 se presenta la definición de la clase *FrameHeaderClass* que el programador debe incluir.

```
static class FrameHeaderClass : public PacketHeaderClass{
public:

/* Constructor que invocará al constructor de su clase
 * base(PacketHeaderClass) para registrar la estructura que
 * representa a la trama en el espacio OTcl
 */
    FrameHeaderClass() : PacketHeaderClass("PacketHeader/Frame",
sizeof(hdr_frame)){

/* Función que permite saber donde estará almacenado offset_ de
 * hdr_frame
 */
          bind_offset(&hdr_frame::offset_);
    }
}class_framehdr;// Definición de un objeto estático de la clase
                // FrameHeaderClass
```

Código 3.4 Definición de la clase *FrameHeaderClass*.

1.7.1.1.2. Agentes

Como se explicó anteriormente, los protocolos son representados mediante agentes, los cuales son considerados como entidades de procesamiento. Los agentes consisten en objetos de la clase *Agent* (o de alguna clase derivada de *Agent*), en donde se implementa la funcionalidad para la creación, procesamiento y destrucción de la unidad de datos **[12]**.

De acuerdo a la funcionalidad del protocolo "*simplex* sin restricciones", se ha considerado implementar dos clases derivadas de la clase *Agent*. La clase *Protocolo1SenderAgent* que representa al agente emisor y la clase *Protocolo1ReceiverAgent* que representa al agente receptor a nivel de la capa de enlace de datos.

Clase Protocolo1SenderAgent

El objetivo de esta clase es enviar tramas al receptor sin ninguna limitación, por tanto la clase provee funcionalidad para: obtener un paquete de la capa de red, construir una trama y enviar la trama a su destino.

En el Código Protocolo 1.1 se presenta la definición de la clase *Protocolo1SenderAgent*, en donde se presentan las variables y funciones que son

propias para la representación del protocolo, como también aquellas que son necesarias para la simulación en NS-2.

Cabe mencionar que tanto los nombres de variables como de funciones se encuentran de acuerdo a la nomenclatura de la referencia **[7]**.

```
#include <stdlib.h>
#include "agent.h"
#include "frame.h"

class Protocolo1SenderAgent: public Agent{
private:

// Variables del protocolo
     frame  *s_;             // Variable que representa al trama
                             // que será enviada
     paquete buffer_;        // Paquete que se obtiene de la
                             // capa de red
     Packet *p_s_;           // Variable con la que se podrá
                             // acceder a frame(hdr_frame)
     int frameSize_;         // Tamaño de la trama
     int npaquete_;          // Representa el número de paquetes
                             // que se obtendrá de la capa de red
public:
// Constructor
     Protocolo1SenderAgent();
// Destructor
     ~Protocolo1SenderAgent();

// Funciones del protocolo
     void from_network_layer(paquete * msg);   // Obtiene un paquete
                                               // de la capa de red

     void to_physical_layer(Packet * envio);   // Envía una trama
                                               // hacia la capa física

     void initialize();                        // Empieza con la
                                               // transmisión de datos

// Función que se redefine de la clase Agent y será invocada
// indirectamente desde el script de simulación OTcl
     int command (int argc, const char*const* argv);

// Funciones auxiliares
     void crear_paquete(paquete *msg);  // Genera un paquete en la
                                        // capa de red

     void registro_evento();            // Función que invoca a un
                                        // procedimiento en OTcl para
                                        // imprimir en pantalla el
                                        // nombre del nodo y el
                                        // tiempo que ocurre un
                                        // evento en éste
};
```

Código Protocolo 1.1 Definición de la clase *Protocolo1SenderAgent*.

Variables

- **s_:** Variable que representa la trama que será enviada.
- **buffer_:** Variable que representa la información obtenida de la capa red (paquete), la cual será insertada dentro del campo de datos de la trama.
- **p_s_:** Variable de tipo *Packet* con la cual se podrá acceder a la trama de salida para modificar sus campos.
- **frameSize_:** Representa el tamaño trama, este valor deberá ser configurado por el usuario en el *script* de simulación**.**
- **npaquete_:** Variable con la que se configura el límite al flujo de tramas. Su uso es sólo para efectos de la simulación pues la naturaleza del protocolo es enviar un flujo infinito de tramas.

Funciones

- **Protocolo1SenderAgent():** Constructor en donde se establecerá el enlace bidireccional entre los espacios C++ y OTcl para las variables *frameSize_* y *npaquete_* (mediante la función *bind()* de la clase *TclObject*), de tal manera que el valor configurado desde el espacio OTcl se vea reflejado en el espacio C++ y viceversa.

  Además se pasará como argumento al constructor de la clase *Agent* el valor *PT_FRAME,* para identificar el tipo de unidad de datos que procesará el agente implementado por esta clase.

Su implementación se presenta en el Código Protocolo 1.2.

```
Protocolo1SenderAgent::Protocolo1SenderAgent() : Agent(PT_FRAME)
{

/* Enlace bidireccional de las variables frameSize_ y npaquete_
 * entre los espacio C++ y OTcl, cabe mencionar que no es necesario
 * que la variable tenga el mismo nombre en los dos espacios
 */
    bind("frameSize_",&frameSize_);
    bind("npaquete_",&npaquete_);
}
```

Código Protocolo 1.2 Implementación del constructor de la clase *Protocolo1SenderAgent.*

- **~Protocolo1SenderAgent():** Destructor en donde se libera el espacio de memoria reservada para los objetos que has sido creados dinámicamente por el operador *new* a través del operador *delete.* En el Código Protocolo 1.3 se presenta su implementación.

```
Protocolo1SenderAgent::~Protocolo1SenderAgent()
{

// Se libera el espacio  de memoria asignado a
// al puntero s_ que representa a la trama que se
// enviará
     delete s_;
}
```

Código Protocolo 1.3 Implementación del destructor de la clase *Protocolo1SenderAgent.*

- **from_network_layer():** Función que en conjunto con *to_network_layer()* (función de la clase *Protocolo1ReceiverAgent*) representan la interfaz entre la capa de red y la capa de enlace de datos. En esta función la capa de enlace de datos obtiene un paquete de la capa de red mediante la invocación a la función *crear_paquete().* Su implementación se presenta en Código Protocolo1.4.

```
void Protocolo1SenderAgent::from_network_layer(paquete *msg)
{
     printf(" La capa de red envía un paquete \n");
     crear_paquete(msg);
}

void Protocolo1SenderAgent::crear_paquete(paquete * msg)
{      // Creación de un nuevo paquete
         msg->data=" Nuevo Paquete";

}
```

Código Protocolo 1.4 Implementación de la función *from_network_layer()* y *crear_paquete()* de la clase *Protocolo1SenderAgent.*

- **to_physical_layer():** Función que en conjunto con la función *recv()* (función de la clase *Protocolo1ReceiverAgent*) representan la interfaz entre la capa de enlace de datos y la capa física, en donde la capa de enlace de datos envía una trama a la capa física. Para su implementación se ha utilizado la función *send()* definida en la clase *Agent* cuya finalidad es enviar *psend.* Su implementación se presenta en Código Protocolo 1.5.

```
void Protocolo1SenderAgent::to_physical_layer(Packet *psend)
{
     printf(" Envía una nueva trama a la capa física\n");
     // Función definida en la clase Agent
     send(psend,0);
}
```

Código Protocolo 1.5 Implementación de la función *to_physical_layer()* de la clase *Protocolo1SenderAgent.*

- **command():** Función redefinida de la clase *Agent*, y que será invocada indirectamente desde el *script* de simulación OTcl. Su implementación se presenta en el Código Protocolo 1.6

  El argumento *argv[0]* contiene el nombre del comando *cmd*, y *argv[1]* es la operación solicitada (*iniciar_transmision*). Dentro de esta función se insertar el código que realice la operación deseada, es así que para el caso en el que usuario ingrese *iniciar_transmision* se invocará a la función *initialize().*

```
// Función que será invocada indirectamente desde el
// script de simulación OTcl
int Protocolo1SenderAgent::command(int argc, const char*const* argv)
{
// El valor argc indica el número de elementos que contiene el
     // array argv[]
     //    argv[0]=”cmd”
     //    argv[1]=”iniciar_transmision”
  if (argc == 2) {
    if (strcmp(argv[1], "iniciar_transmision") == 0) {

          // Función definida para empezar
          // con la transmisión
          initialize();
    return (TCL_OK);
    }
  }
return (Agent::command(argc, argv));
}
```

Código Protocolo 1.6 Implementación de la función *command()* de la clase *Protocolo1SenderAgent*.

- **initialize()**: Función que inicia con la transmisión de datos; para este protocolo en ésta se simula una transmisión de datos infinita, por lo que se hace uso de un lazo *for,* en donde, por efectos de visualización de resultados se fija un límite. En el cuerpo del lazo *for* se implementa la funcionalidad de entramado, y además se asigna el valor de la variable

frameSize al campo *size_* de la estructura *hdr_cmn* y finalmente se procede a enviar la trama a la capa física.

En el Código Protocolo 1.7 se presenta la implementación de la función *initialize().*

```
void  Protocolo1SenderAgent::initialize()
{
    for(int i=0;i<npaquete_;i++)
    {
          // Se imprime en en pantalla
          // el nombre del nodo y el tiempo
          registro_evento();

          // Reserva espacio de memoria para el puntero p_s_ y
          // además configura los campos de la estructura hdr_cmn
          p_s_=allocpkt();

          // Con el Packet p_s_ se accede a frame(hdr_frame)
          s_ = frame::access(p_s_);

          // Se obtiene un paquete de la capa de red
          from_network_layer(&buffer_);

          // Se coloca el paquete en el campo info de la trama
          s_->info_var=buffer_;

          // Se accede a la cabecera hdr_cmn
          hdr_cmn *ch = hdr_cmn::access(p_s_);

          // Se asigna el tamaño de la trama de acuerdo al valor
          // configurado en el espacio OTcl
          ch->size()=frameSize_;

          // Se envía la trama a la capa física
          to_physical_layer(p_s_);

          printf(" ...Esperando un nuevo paquete\n");
    }
}
```

Código Protocolo 1.7 Implementación de la función *initialize()* de la clase *Protocolo1SenderAgent*.

Para implementar la funcionalidad de entramado, es necesaria la utilización de funciones de las clases *Agent* y *Packet*.

La función *allocpkt()* de la clase *Agent* reserva el espacio de memoria para el puntero *p_s_* (objeto de la clase *Packet),* configura valores por defecto para los campos de la estructura *hdr_cmn*, tales como *ptype_*, *size_*, *uid_*, *ts_* y otros que son relevantes en esta implementación.

Con el puntero *p_s_* se ingresa a la cabecera que representa a *frame (hdr_frame)* mediante la función *access(),* para posteriormente colocar el paquete recibido de la capa de red (mediante la función *from_network_layer()*) en el campo correspondiente a los datos *(s−>info_var=buffer).*

- **crear_paquete():** Función que crea un paquete en la capa de red. En el Código Protocolo 1.4 se presenta la respectiva implementación.
- **registro_evento():** Función que es invocada para imprimir el nombre del nodo y el tiempo en que ocurre un evento sobre el mismo. Esta función enviará un *string* (*out*) al intérprete OTcl, el cual contiene el comando que hará la invocación al procedimiento *registro_evento{}* implementado en el *script* OTcl (se explicará en la Sección 3.1.1.2). La implementación se presenta en Código Protocolo 1.8.

```
void Protocolo1SenderAgent::registro_evento()
{
     char out[100];

     // Se prepara un string que será enviado al intérprete OTcl
     sprintf(out, "%s registro_evento %3.1f",
    name(),(Scheduler::instance().clock())*1000);

     // Se obtiene una instancia del intérprete
     Tcl& tcl = Tcl::instance();

     // Se envía un string de salida para el intérprete OTcl
     tcl.eval(out);

}
```

```
# Función para imprimir el nombre del nodo y el tiempo
Agent/Protocolo1Sender instproc registro_evento {time} {
        $self instvar node_
        puts "-----------------------------------"
        puts "NODO:[$node_ id], TIME: $time ms"
}
```

Código Protocolo 1.8 Implementación de la función *registro_evento()* y del procedimiento *registro_evento{}* para la clase *Protocolo1SenderAgent*.

Clase Protocolo1ReceiverAgent

En el Código Protocolo 1.9 se presenta la definición de la clase *Protocolo1ReceiverAgent*.

```
class Protocolo1ReceiverAgent: public Agent{

private:

// Variables del protocolo
     frame *r_;              // Variable que representa a la trama
                             // que será recibida

public:
// Constructor
     Protocolo1ReceiverAgent();

// Destructor
     ~Protocolo1ReceiverAgent();

// Función del protocolo
     void to_network_layer(paquete *msg);      // Envía un paquete a
                                               // la capa de red
// Función de la clase Agent que se redefine
     void recv(Packet* pkt, Handler*);         // Función que recibe
                                               // a la trama desde la
                                               // capa física
// Función auxiliar
     void registro_evento(); // Función que invoca a un
                             // procedimiento en OTcl para
                             // imprimir en pantalla
                             // el nombre del nodo y el
                             // tiempo que ocurre un evento en éste
};
```

Código Protocolo 1.9 Definición de la clase *Protocolo1ReceiverAgent*.

Variable

- **r_:** Representa a la trama que se recibirá desde la capa física.

Funciones

Tanto el constructor como el destructor de la clase, tienen implementación similar a los respectivos de la clase *Protocolo1SenderAgent*, por tanto no se considera necesaria su explicación.

- **to_network_layer():** Función que permite enviar a la capa de red el paquete recibido en el campo de datos de la trama. En Código Protocolo 1.10 se presenta la implementación de la función.

```
void Protocolo1ReceiverAgent::to_network_layer(paquete* msg)
{
          // Se obtiene los datos del paquete recibido
          printf(" La capa de red recibe: %s\n",msg->data);
}
```

Código Protocolo 1.10 Implementación de la función *to_network_layer()* de la clase *Protocolo1ReceiverAgent*.

- **recv():** Función que es invocada cada vez que el agente recibe un objeto de tipo *Packet*. Con el objeto recibido (*p_r_*) se accede a *frame* para desencapsular el paquete contenido en el campo *info_var* y se procede a eliminar el objeto *p_r_* recibido. En el Código Protocolo 1.11 presenta la implementación de la función.

```
void Protocolo1ReceiverAgent::recv(Packet* p_r_, Handler*)
{    // Función que imprimirá en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();

     // Con p_r_ se accede a frame
     r_ = frame::access(p_r_);

     // Se envía los datos hacia la capa de red
     to_network_layer(&r_->info_var);

     // Se libera el espacio de memoria
     delete p_r_;
}
```

Código Protocolo 1.11 Implementación de la función *recv()* de la clase *Protocolo1ReceiverAgent*.

Clase Protocolo1SenderClass

Clase que será utilizada en el proceso de vinculación entre los espacios C++ y OTcl; de esta clase se instancia el objeto *class_protocolo1_sender*, el cual vinculará un objeto de la clase *Agent/Protocolo1Sender* (clase creada en el tiempo de ejecución en el espacio OTcl) con su objeto reflejo en C++ (objeto de la clase *Protocolo1SenderAgent*) **[13]**. El proceso de vinculación se explicará con detalle en la Sección 3.3.1.

En el Código Protocolo 1.12 se presenta la definición de la clase *Protocolo1SenderClass* .

```
static class Protocolo1SenderClass: public TclClass{
public:
/* Se invoca al constructor de la clase TclClass, enviando como
 * argumento el nombre Agent/Protocolo1Sender con el que se hará
 * referencia a la clase Protocolo1SenderAgent en el espacio OTcl
 */
     Protocolo1SenderClass(): TclClass("Agent/Protocolo1Sender"){}
     TclObject * create (int, const char*const*){
     return (new Protocolo1SenderAgent());
     }
} class_protocolo1_sender;// Objeto estático de la clase
                          // Protocolo1SenderClass
```

Código Protocolo 1.12 Definición de la clase *Protocolo1SenderClass.*

Para el proceso de vinculación de la clase *Protocolo1ReceiverAgent*, se ha implementado una clase similar a la antes descrita con el nombre de *Protocolo1ReceiverClass*, por lo que su explicación no se ha considerado necesaria.

En la Figura 3.6 se presenta gráficamente la funcionalidad para el protocolo "*símplex* sin restricciones".

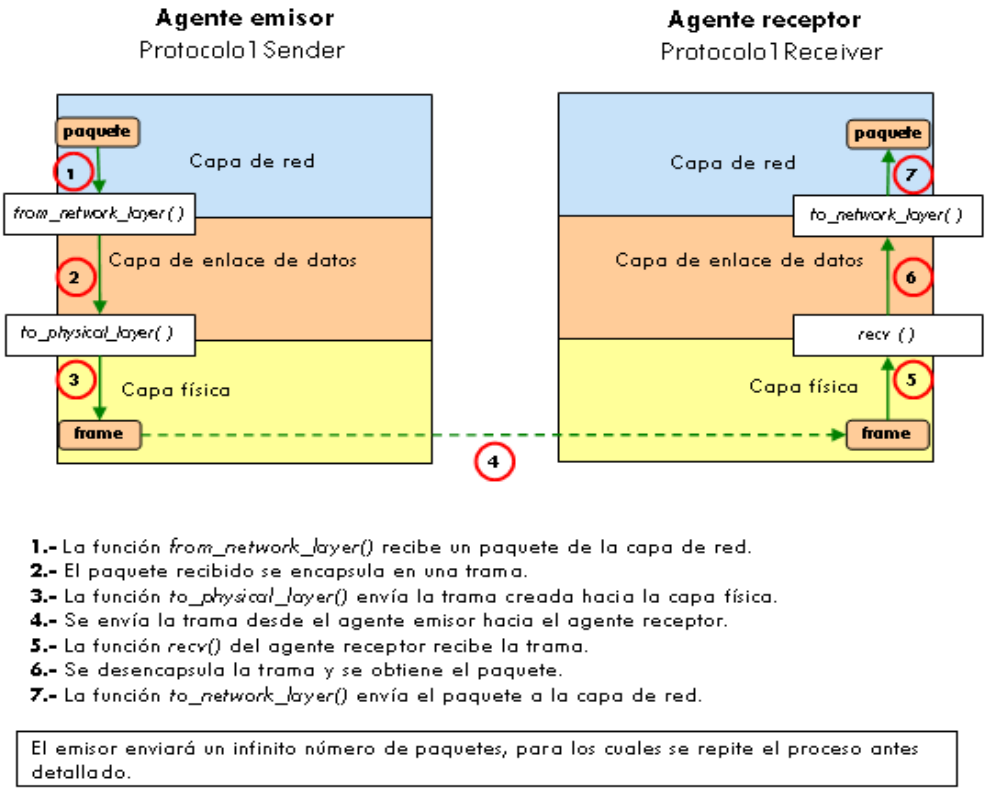


Figura 3. 6 Representación gráfica del funcionamiento del protocolo "*símplex* sin restricciones".

1.7.1.2. Configuración del escenario de simulación

```
#=========================================================#
# Parámetros que ingresará el usuario
#=========================================================#
set variable(propagacion_)    50     ;# Tiempo de propagación en el
                                     ;# canal en ms
set variable(vtx_)            10     ;# Velocidad de transmisión en Mbps
set variable(frameSize_)      1000   ;# Tamaño de la trama en bytes
set variable(npaquete_)       10     ;# Número de paquetes que se
                                     ;# obtendrá desde la capa de red
set variable(tr)          out1.tr    ;# Archivo de trazas .tr
set variable(nam)         out1.nam   ;# Archivo de trazas .nam
set variable(iniciar)         0      ;# Inicio de la transmisión
set variable(detener)         1.0    ;# Fin de la simulación
#=========================================================#
```

```
# Se crea un objeto de la clase Simulator
set ns [new Simulator]

# Diferentes colores para los flujos de datos
$ns color 1 Blue
$ns color 2 Red

# Creación del archivo de trazas .nam
set nf [open $variable(nam) w]
$ns namtrace-all $nf

# Creación del archivo de trazas .tr
set ntr [open $variable(tr) w]
$ns trace-all $ntr

# Definición del procedimiento finish que es invocado para
# finalizar la simulación
proc finish {} {
      global ns nf ntr
      $ns flush-trace
      close $nf
      close $ntr
      exit 0
}

# Procedimiento para imprimir en pantalla el nombre del nodo
# y el tiempo en el que ocurre un evento
Agent/Protocolo1Sender instproc registro_evento {time} {
      $self instvar node_
      puts "-----------------------------------"
      puts "NODO:[$node_ id], TIME: $time ms"
}

# Procedimiento para imprimir el nombre del nodo y el tiempo en el
# que ocurre un evento
Agent/Protocolo1Receiver instproc registro_evento {time} {
      $self instvar node_
      puts "-----------------------------------"
      puts "NODO:[$node_ id], TIME: $time ms"
}


# Creación de los nodos emisor(n0) y receptor(n1)
set n0 [$ns node]
set n1 [$ns node]

# Configuración del color de los nodos
$n0 color blue
$n1 color red

# Creación del canal que enlaza los nodos
$ns simplex-link $n0 $n1 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail

# Creación del agente para el nodo emisor
set A [new Agent/Protocolo1Sender]
$A set frameSize_ $variable(frameSize_)
$A set tamaniobuffer_ $variable(tamaniobuffer_)
```

```
# Creación del agente para el nodo receptor
set B [new Agent/Protocolo1Receiver]
# Asignación de los agentes a los nodos
$ns attach-agent $n0 $A
$ns attach-agent $n1 $B
# Conexión entre los agentes
$ns connect $A $B

# Se define el color a los flujos de datos
$A set fid_ 1
$B set fid_ 2

# Inicio de la transmisión
$ns at $variable(iniciar) "$A iniciar_transmision "

# Fin de la simulación
$ns at $variable(detener) "finish"
$ns run
```

Script Protocolo1 Configuración del escenario para simular el protocolo "*simplex* sin restricciones".

La configuración del escenario para el protocolo *"simplex* sin restricciones" se la ha realizado en un *script* de simulación, el cual consiste en un archivo con extensión .tcl. A continuación, se describen los comandos que se han utilizado para la configuración del *script*.

Inicialmente se han establecido los valores que deberán ser configurados por el usuario, tales como tiempo de propagación en el canal, velocidad de transmisión, tamaño de la trama, etc. Para la asignación de valores de los parámetros se hace uso de *arrays* como por ejemplo *variable*(*frameSize_*), cabe indicar que en los *arrays* en OTcl cualquier cadena puede ser el índice del *array*, en este caso el *array* es "variable" y *frameSize_* es el índice de dicho *array*, el valor asignado al índice del array es 50.

```
#=========================================================#
# Parámetros que ingresará el usuario
#=========================================================#
set variable(propagacion_)    50      ;# Tiempo de propagación en el canal
                                      ;# en ms
set variable(vtx_)            10      ;# Velocidad de transmisión en Mbps
set variable(frameSize_)      1000    ;# Tamaño de la trama en bytes
set variable(npaquete_)       10      ;# Número de paquetes que se
                                      ;# obtendrá desde la capa de red
set variable(tr)           out1.tr    ;# Archivo de trazas .tr
set variable(nam)          out1.nam   ;# Archivo de trazas .nam
set variable(iniciar)         0       ;# Inicio de la transmisión
set variable(detener)         1.0     ;# Fin de la simulación
#=========================================================#
```

Para empezar la simulación es necesario crear un objeto de la clase S*imulator,* que en este caso se lo ha llamado *ns,* con éste se podrá invocar a los procedimientos de esta clase **[13]**.

```
set ns [new Simulator]
```

Se abre un archivo out1.nam con el comando *open* el mismo que retorna un identificador que el es asignado a la variable *nf*, del mismo modo se abrirá un archivo out1.tr cuyo identificador será asignado a la variable *ntr*.

```
set nf [open $opt(nam) w]
$ns namtrace-all $nf

set ntr [open $opt(tr) w]
$ns trace-all $ntr
```

Con el objeto *ns* se invoca al procedimiento *namtrace-all{}* para indicar al simulador que grabe las trazas de simulación en un formato que sea entendible para la aplicación NAM. De manera similar procede el procedimiento *trace-all{}* que grabará las trazas de los eventos en un formato general.

Posteriormente se define un procedimiento *finish{}* en donde con el objeto *ns* se invoca al procedimiento *flush-trace{}* el cual almacena las trazas registradas en los archivos que se abrieron y posteriormente se cierran los archivos (out.nam y out.tr) **[5]**.

```
proc finish {} {
      global ns nf ntr
      $ns flush-trace
      close $nf
      close $ntr
      exit 0
}
```

Se ha definido un procedimiento *registro_evento{}* para la clase *Agent/Protocolo1Sender* el cual es invocado desde la función *registro_evento()* implementada en el espacio C++ como se explicó anteriormente. Su funcionalidad consiste en imprimir el identificador del nodo en el espacio OTcl, y el tiempo en el que ésta es invocada[1]. Para la clase *Agent/Protocolo1Receiver* se ha implementado un procedimiento con el mismo nombre y funcionalidad que la anteriormente descrita.

[1] El procedimiento *registro_evento{}* es invocado cada vez que ocurre un evento en el nodo al cual esta asignado.

```
# Desde el espacio C++ se envía como argumento el time
Agent/Protocolo1Sender instproc registro_evento {time} {
      # Declaración de la variable node_, la cual es heredada de
      # su  clase base
      $self instvar node_
      puts "-----------------------------------"
      puts "NODO:[$node_ id], TIME: $time ms"
}
```

Para la definición de la topología física, se han creado dos nodos que representan al emisor y receptor, respectivamente, mediante el procedimiento *node{}* de la clase *Simulator*. La conexión entre los nodos se la hace mediante la creación de un canal *simplex* empleando el procedimiento *simplex-link{}* ya que solo se necesita la transmisión unidireccional (ver la Figura 3.7).

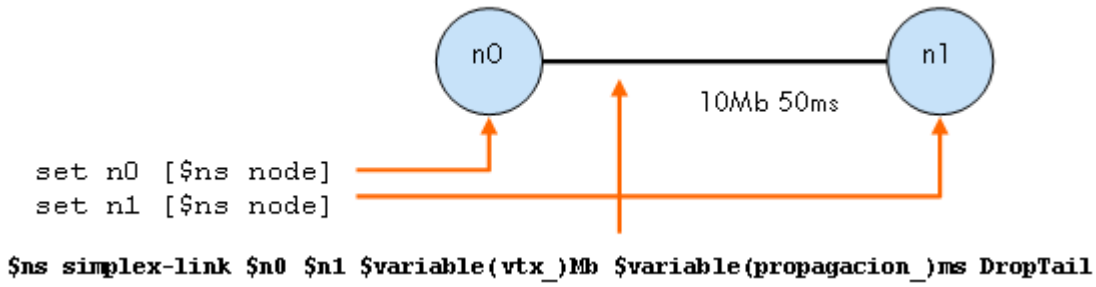


Figura 3.7 Creación de un enlace unidireccional en el *script* OTcl.

El procedimiento *simplex-link{}* tiene como argumentos: los nodos a ser conectados (n0, n1), el ancho de banda (10*Mb*), el retardo (50*ms*) y el tipo de encolamiento (*DropTail*[1]).

A continuación se crea un agente emisor (objeto de la clase *Agent/Protocolo1Sender*) en el cual se configura el tamaño de las tramas que serán transmitidas y el número de paquetes que se obtendrán desde la capa de red; posteriomente se crea el agente receptor (objeto de la clase *Agent/Protocolo1Receiver*),

Los agentes son asignados a los nodos emisor (A) y receptor (B), respectivamente como se indica en la Figura 3.8.

[1] Clase que implementa una cola de tipo FIFO (*First input first output)* de tal manera que cuando el búfer de la cola este lleno los paquetes que llegan se descartan.

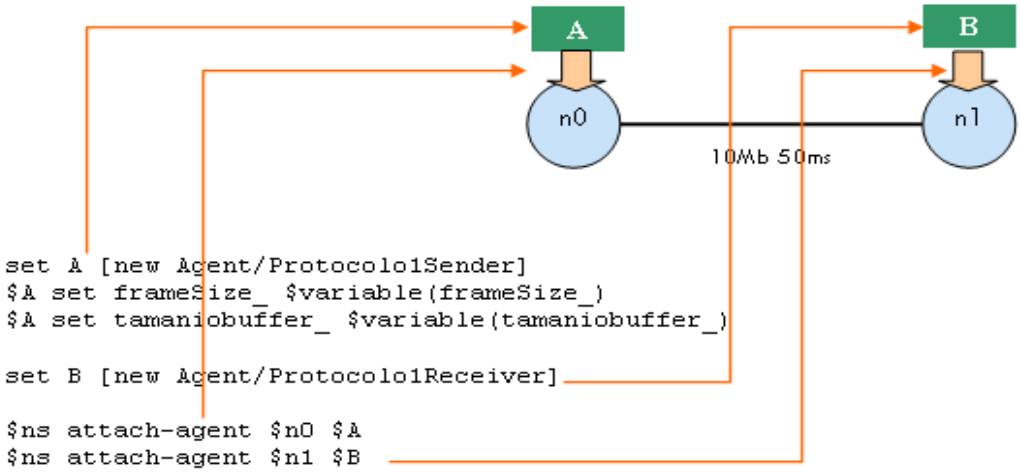


Figura 3.8 Asignación de los agentes a los nodos.

A continuación se establece una conexión lógica entre los agentes mediante el procedimiento *connect{}* de la clase *Simulator* como se indica en la Figura 3.9.

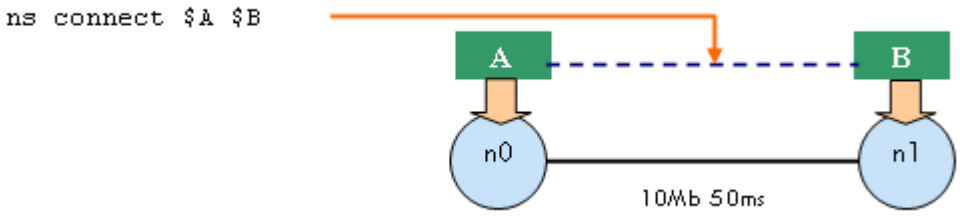


Figura 3.9 Conexión lógica entre los agentes.

Con el procedimiento *at{}* de la clase *Simulator* se realiza la planificación para la ejecución de eventos en un tiempo de simulación dado; de esta manera al tiempo de 0 segundos se indica que el agente A debe empezar con la transmisión de datos, y al tiempo 1 segundo que invoque al procedimiento *finish{}* para finalizar la simulación.

```
# Inicio de la transmisión
$ns at $variable(iniciar) "$A iniciar_transmision "

# Fin de la simulación
$ns at $variable(detener) "finish"
```

Finalmente, para correr la simulación se hace la llamada al procedimiento *run{}* de la clase *Simulator*

```
$ns run
```

1.7.1.3. Compilación y ejecución

Los pasos necesarios para la compilación y ejecución del protocolo implementado son:

1. Ubicar los archivos que contienen la definición e implementación de los protocolos dentro del directorio “ns-2.30” (o dentro de algún subdirectorio de éste). Por ejemplo los archivos que contienen la implementación del protocolo “*símplex* sin restricciones” (protocolo1.h, protocolo1.cc, frame.h, frame.cc y paquete.h) se han ubicado dentro del directorio “protocolo1”, que a su vez se encuentra dentro del directorio “protocoloscapaenlace”.

2. Editar el archivo *Makefile* ubicado en el subdirectorio ns-2.30; en donde se debe agregar el nombre del archivo de salida con extensión .o (nombre que deberá que ser igual al nombre del archivo que contiene la definición) que será creado en tiempo de compilación.

```
OBJ_CC = \
common/scheduler.o common/object.o common/packet.o \
common/ip.o routing/route.o common/connector.o common/ttl.o \
trace/trace.o trace/trace-ip.o \
      .
      .
protocoloscapaenlace/protocolo1/protocolo1sender.o \
protocoloscapaenlace/protocolo1/protocolo1receiver.o \
@V_STLOBJ@
```

3. En el archivo *ns-default.tcl* (su ubicación se presenta en el Anexo B) se debe establecer los valores por defecto de los parámetros configurables del protocolo. En este caso, los parámetros a configurar para el agente emisor son: *frameSize_* y *tamaniobuffer_*.

```
Agent/Protocolo1Sender set frameSize_ 1000
Agent/Protocolo1Sender set tamaniobuffer 10
```

4. En la línea de comandos se debe ubicar en el subdirectorio ns-2.30 para ejecutar el comando *make depend* (solo si se ha realizado modificaciones en el archivo *ns-default.tcl*) y posteriormente *make*.

5. Finalmente, para la ejecución de la simulación, ubicarse en el directorio en el que se encuentre el *script* OTcl que contiene la configuración del escenario del protocolo (protocolo1.tcl) y ejecutar el comando *ns protocolo1.tcl.*

1.7.1.4. Resultados de la simulación

Los resultados de la simulación se los obtiene en los archivos out1.tr y out1.nam, además se puede observar los mensajes que se imprimen en pantalla.

1.7.1.4.1. Impresión en pantalla

A continuación, en la Figura 3.10 se indican los resultados que se imprimen en pantalla al realizar la simulación del protocolo "*simplex* sin restriciones".

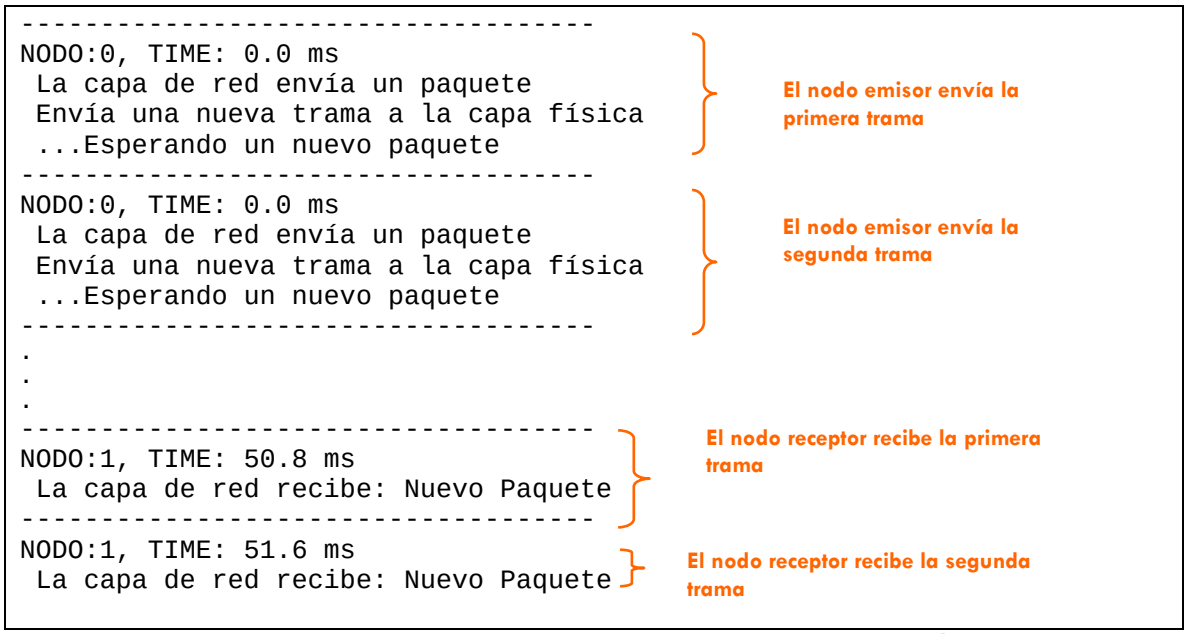


Figura 3.10 Resultados impresos en pantalla obtenidos de la simulación del protocolo "*simplex* sin restricciones".

1.7.1.4.2. Archivo out1.tr

En la Figura 3.11 se presentan parte de los resultados que se registran en el archivo out1.tr.

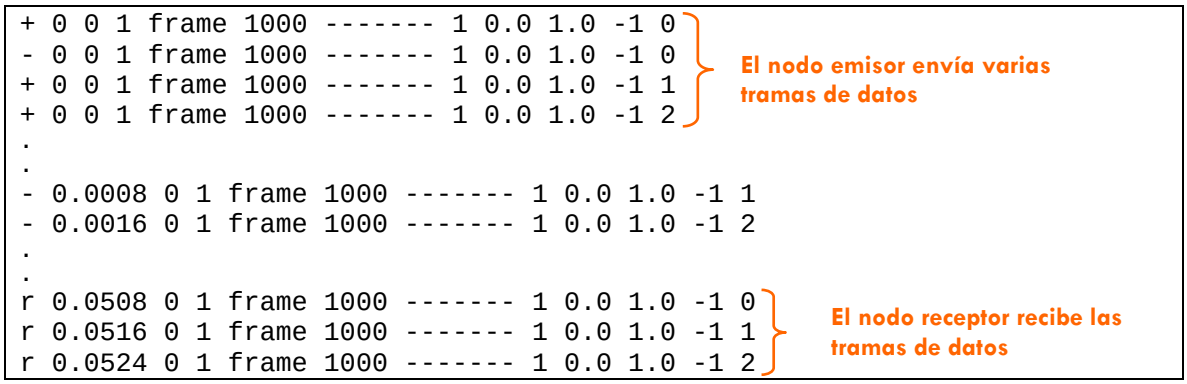


Figura 3.11 Resultados registrados en el archivo out1.tr.

Al observar el contenido de este archivo, se distinguen todos los eventos registrados durante la simulación, uno en cada línea. Es así que en la primera línea del archivo out1.tr representa la primera trama generada cuya interpretación se la muestra en la Figura 3.12.

```
+ 0.0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
```

+	En la cola de espera
0.0	Evento generado al tiempo 0.0 segundos
0	Nodo emisor
1	Nodo receptor
frame	Unidad de datos(*frame*) cuyo nombre fue asignado en la clase *p_info*
1000	Tamaño de la unidad de datos(*frame*) en bytes
-------	Representa las banderas que en este caso no han sido configuradas
1	Identificar del flujo
0.0	Dirección origen, es decir en este caso se ha enviado desde el nodo 0 y puerto 0
1.0	Dirección destino, en este caso se recibirá en el nodo 1 y puerto 0
-1	Valor por defecto del número de secuencia
0	Identificador del paquete

Figura 3.12 Interpretación de la primera línea del archivo out1.tr.

La segunda línea se diferencia de la antes descrita en el primer término (–), el cual representa el desencolamiento al tiempo 0.0 de la primera trama generada.

```
- 0.0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
```

Las siguientes nueve líneas representan las nueve tramas generadas a continuación; únicamente se diferencian de la primera línea en el valor del identificador del paquete.

El desencolamiento de las siguientes nueve tramas no es inmediato como sucedió en para la primera trama, esto dependerá de la velocidad de transmisión configurada (10Mbps), por tanto la segunda trama saldrá de la cola de espera al tiempo 0.0008 segundos y será recibida por el nodo emisor al tiempo 0.0516 segundos, la obtención del valor se la presenta a continuación.

Tiempo de llegada = tiempo de salida + tiempo de transmisión

+ tiempo de propagación en el canal

Tiempo de llegada = 0.0008 s+ (1000*8/1000000)s + 0.050s

Tiempo de llegada = 0.0516 s

1.7.1.4.3. Archivo out1.nam

A continuación, desde la Figura 3.13 hasta la Figura 3.17 se presentan los resultados que se visualizan al ejecutar el archivo out1.nam obtenido de la simulación del protocolo "*simplex* sin restricciones".

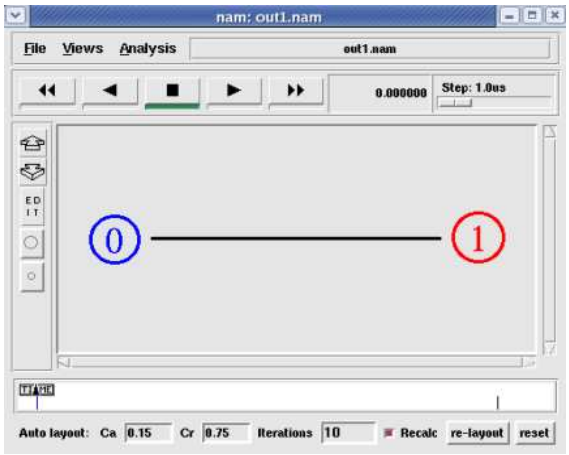


Figura 3.13 Transmisión de la primera trama al tiempo 0 segundos.

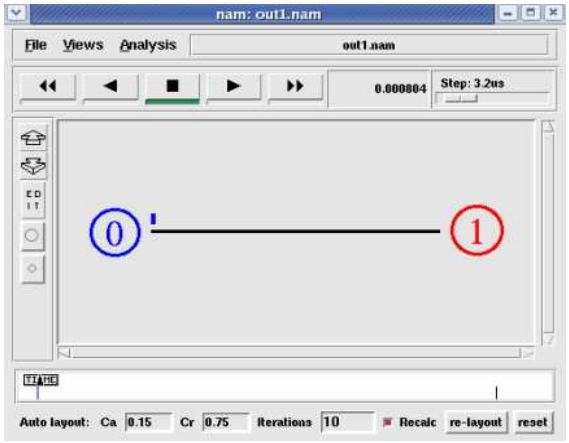


Figura 3.14 Transmisión de la segunda trama.

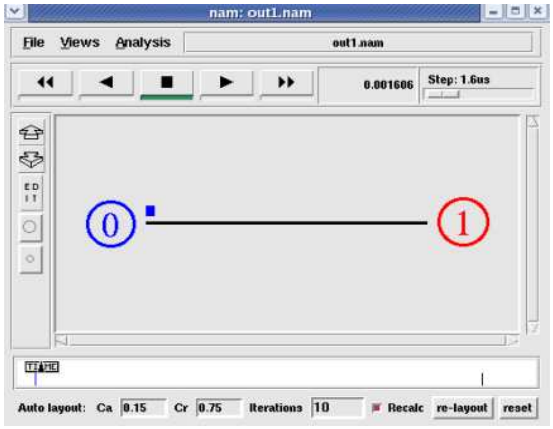


Figura 3.15 Finalización de la transmisión de la segunda trama.

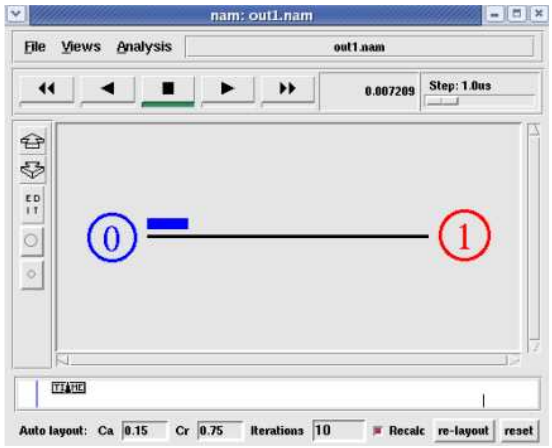


Figura 3.16 Finalización de la transmisión de la última trama.

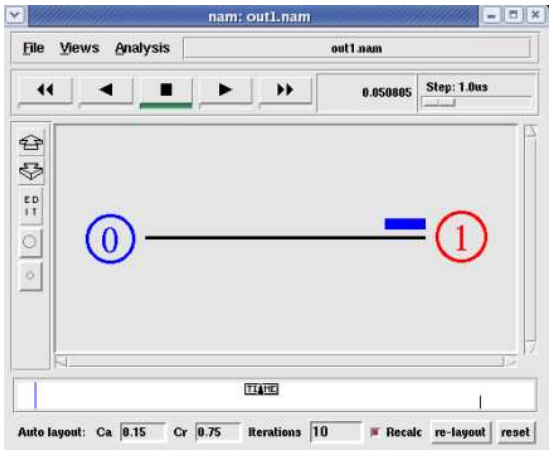


Figura 3.17 Llegada de la primera trama al nodo receptor.

De los resultados obtenidos se puede concluir que el protocolo cumple con las condiciones descritas para el mismo, específicamente con la generación infinita de datos. De esta manera se puede observar que las tramas llegan al módulo receptor de acuerdo al tiempo calculado (tiempo de propagación + tiempo de transmisión), por ejemplo la primera trama llega a tiempo 50.8ms y cada 0.8ms

después llegan las siguientes tramas, además se puede observar que cada agente cumple con su rol de emisor y receptor respectivamente.

1.7.2. PROTOCOLO SIMPLEX DE PARADA Y ESPERA

1.7.2.1. Diseño e implementación en C++

El problema que surge en el protocolo debido a la limitación de búfer y de procesamiento, se ha solucionado con la implementación de control de flujo, en el que el receptor debe enviar una trama de confirmación (acuse de recibo) que informe al emisor que puede continuar con la transmisión de datos.

Tomando en cuenta la consideración de transmisión unidireccional de datos, se requiere implementar la clase *Protocolo2SenderAgent* que represente al emisor y la clase *Protocolo2ReceiverAgent* que represente al receptor.

1.7.2.1.1. Agentes

Clase Protocolo2SenderAgent

El objetivo de esta clase es enviar la trama de datos al receptor y esperar por un trama de confirmación de éste, para continuar con la transmisión de una nueva trama de datos, esto se lo implementa mediante las funciones: *initialize()* y *recv().*

La definición de la clase *Protocolo2SenderAgent* se presenta en el Código Protocolo 2.1.

```
class Protocolo2SenderAgent: public Agent{
private:

// Variables del protocolo
     frame  *s_;       // Variable que representa a la trama
                       // que será enviada
     paquete buffer_;  // Paquete que se obtiene  de la
                       // capa de red
     Packet *p_s_;     // Variable con la que se podrá
                       // acceder a frame(hdr_frame)
     int frameSize_;   // Tamaño de la trama

public:
// Constructor
     Protocolo2SenderAgent();

// Destructor
     ~Protocolo2SenderAgent();

// Funciones  del protocolo
```

```
    void from_network_layer(paquete * msg);   // Obtiene un paquete
                                              // de la capa de red
    void to_physical_layer(Packet * psend);   // Envía una trama
                                              // hacia la capa física
    void initialize();                        // Función que será
                                              // invocada para empezar
                                              // con la transmisión
// Funciones que se redefinen de la clase Agent
    int command (int argc, const char*const* argv);
    void recv(Packet* p_r_, Handler*);  // Recibe la trama desde la
                                        // capa física
// Funciones auxiliares
    void crear_paquete(paquete * msg);  // Genera un paquete en
                                        // la capa de red
    void registro_evento();       // Función que invoca a un
                                  // procedimiento en OTcl para
                                  // imprimir en pantalla el nombre del
                                  // nodo al cual está asignado éste
                                  // agente
};
```

Código Protocolo 2.1 Definición de la clase *Protocolo2SenderAgent*.

Funciones

- **initialize():** Función que será invocada para empezar con la transmisión de datos, es decir únicamente enviará la primera trama de datos debido a que esta función es invocada indirectamente desde el script de simulación y en ella se hace la planificación del primer evento. En el Código Protocolo 2.2 se presenta la implementación de la función.

```
void  Protocolo2SenderAgent::initialize()
{
    // Función que imprimirá en pantalla
    // el nombre del nodo y el tiempo
    registro_evento();
    p_s_=allocpkt();

    // Con el Packet p_s_ se accede a frame(hdr_frame)
    s_ = frame::access(p_s_);

    // Se obtiene un paquete de la capa de red
    from_network_layer(&buffer_);
    // Se coloca el paquete en el campo info de la trama
    s_->info_var=buffer_;
    // Se accede a la cabecera hdr_cmn
    hdr_cmn *ch = hdr_cmn::access(p_s_);
    // Se asigna el tamaño de la trama
    ch->size()=frameSize_;
    // Se pasa la trama a la capa física
    to_physical_layer(p_s_);
    printf(" ...Esperando la confirmación de recepción\n");
}
```

Código Protocolo 2.2 Implementación de la función *initialize()*.

- **recv():** Esta función se ha implementado considerando que se necesita recibir una confirmación por parte del receptor para continuar con la transmisión; una vez que reciba la trama de confirmación la capa de enlace de datos podrá recibir un nuevo paquete de la capa de red, realizar la función de entramado y enviarla a la capa física, finalmente, se borra la trama de confirmación que fue enviada por el receptor (esto ocurre debido a que desde el punto de vista de C++ el objeto es el mismo que se creó en el módulo receptor). En el Código Protocolo 2.3 se presenta la implementación de la función.

```
void Protocolo2SenderAgent::recv(Packet* p_r_, Handler*)
{
     // Función que imprimirá en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();
     printf(" Recibe la trama de confirmación\n");
     p_s_=allocpkt();

     // Con el Packet p_s_ se accede a frame(hdr_frame)
     s_ = frame::access(p_s_);

     // Se obtiene un paquete de la capa de red
     from_network_layer(&buffer_);

     // Se coloca el paquete en el campo info de la trama
     s_->info_var=buffer_;

     // Se accede a la cabecera hdr_cmn
     hdr_cmn *ch = hdr_cmn::access(p_s_);

     // Se asigna el tamaño de la trama de acuerdo al valor
     // configurado en el espacio OTcl
     ch->size()=frameSize_;

     // Se envía la trama a la capa física
     to_physical_layer(p_s_);

     // Borra la trama de confirmación recibida
     delete p_r_;
     printf(" ...Esperando la confirmación de recepción\n");
}
```

Código Protocolo 2.3 Implementación de la función *recv()* de la clase *Protocolo2SenderAgent*.

El cuerpo de la función *recv()* e *initialize()* son similares, la diferencia entre estas funciones radica en que *recv()* es invocada cada vez que llega una trama objeto al nodo, mientras que *initialize()* es invocada sólo para iniciar la transmisión.

Las funciones restantes son similares a las presentadas en el protocolo "*simplex* sin restricciones".

Clase Protocolo2ReceiverAgent

El objetivo de esta clase es recibir la trama de datos, extraer el paquete del campo de datos de la trama, enviarlo a la capa de red, y posteriormente enviar una trama de confirmación; todo este proceso se lo realiza en la función *recv().* La definición de la clase *Protocolo2ReceiverAgent* es presentada en el Código Protocolo 2.4.

```
class Protocolo2ReceiverAgent: public Agent{

private:

// Variables del protocolo
     frame *r_;              // Variable que representa a la
                             // trama que será recibida
     Packet *p_s_;           // Variable que representa a la trama de
                             // confirmación que enviará el receptor
     int frameSize_;         // Tamaño de la trama

public:
// Constructor
     Protocolo2ReceiverAgent();

// Destructor
     ~Protocolo2ReceiverAgent();

// Funciones del protocolo
     void to_network_layer(paquete *msg);      // Envía un paquete a
                                               // la capa de red

     void to_physical_layer(Packet * psend);   // Envía una trama
                                               // hacia la capa física

// Función que se redefine de la clase Agent
     void recv(Packet* p_r_, Handler*);        // Recibe la trama desde
                                               // la capa física

// Función auxiliar
     void registro_evento();       // Función que invoca un
                                   // procedimiento OTcl para
                                   // imprimir en pantalla el nombre del
                                   // nodo y el tiempo en que ocurre un
                                   // evento en éste
};
```

Código Protocolo 2.4 Definición de la clase *Protocolo2ReceiverAgent*.

Variables

- **p_s_:** Objeto de la clase *Packet* que será utilizado para acceder a la cabecera *hdr_cmn* y configurar el tamaño para la trama de confirmación que se enviará al emisor.

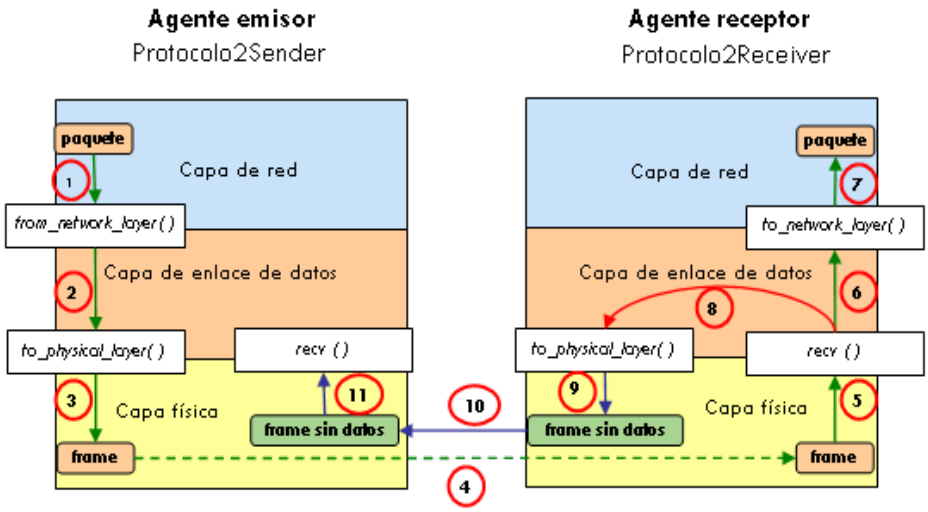
Funciones

- **recv():** Función en la que después de recibir la trama de datos y procesarla, envía una trama de confirmación para cumplir con la función de retroalimentación. Su implementación es presentada en el Código Protocolo 2.5.

```
void Protocolo2ReceiverAgent::recv(Packet* p_r_, Handler*)
{
     // Función que imprimirá en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();

     // Con p_r_ se accede a frame
     r_ = frame::access(p_r_);
     printf(" Recibe la trama de datos\n");

     // Se envía los datos hacia la capa de red
     to_network_layer(&r_->info_var);

     // Se crea una nueva trama para enviar como confirmación
     p_s_=allocpkt();
     hdr_cmn *ch = hdr_cmn::access(p_s_);

     // Se asigna el tamaño a la trama de confirmación
     ch->size()=10;

     // Se envía la trama a la capa física
     to_physical_layer(p_s_);

     // Se borra la trama recibida
     delete p_r_;
}
```

Código Protocolo 2.5 Implementación de la función *recv()* de la clase *Protocolo2ReceiverAgent*.

A continuación, en la Figura 3.18 se presenta el proceso que desencadenan los agentes emisor y receptor que son representados por las clases *Protocolo2Sender* y *Protocolo2Receiver,* respectivamente.



1.- La función *from_network_layer()* recibe un paquete de la capa de red.
2.- El paquete recibido se encapsula en una trama.
3.- La función *to_physical_layer()* envía la trama creada hacia la capa física.
4.- Se envía la trama desde en agente emisor hacia el agente receptor.
5.- La función *recv()* del agente receptor recibe la trama.
6.- Se desencapsula la trama y se obtiene el paquete.
7.- La función *to_network_layer()* envía el paquete a la capa de red.
8.- Se crea una trama de confirmación que se enviará de retorno.
9.- La función *to_physical_layer()* envía la trama de confirmación hacia la capa física.
10.- Se envía la trama de confirmación desde el agente receptor hacia el agente emisor.
Posteriormente el emisor enviará un nuevo paquete y se repetirá el proceso antes descrito.

Figura 3.18 Representación gráfica del funcionamiento del protocolo "*simplex* de parada y espera".

1.7.2.2. Configuración del escenario de simulación

```
#=========================================================#
# Parámetros que ingresará el usuario
#=========================================================#
set variable(propagacion_)  50     ;# Tiempo de propagación en el
                                   ;# canal en ms
set variable(vtx_)          10     ;# Velocidad de transmisión en Mbps
set variable(frameSize_)    1000   ;# Tamaño de la trama en bytes
set variable(tr)          out2.tr  ;# Archivo de trazas .tr
set variable(nam)         out2.nam ;# Archivo de trazas .nam
set variable(iniciar)       0      ;# Inicio de la transmisión
set variable(detener)       1.0    ;# Fin de la simulación
#=========================================================#

# Se crea un objeto de la clase Simulator
set ns [new Simulator]

# Diferentes colores para los flujos de dato
$ns color 1 Blue
$ns color 2 Red

# Creación del archivo de trazas .nam
set nf [open $variable(nam) w]
$ns namtrace-all $nf
```

```
# Creación del archivo de trazas .tr
set ntr [open $variable(tr) w]
$ns trace-all $ntr

# Definición del procedimiento finish que es invocado para
# finalizar la simulación
proc finish {} {
     global ns nf ntr
     $ns flush-trace
     close $nf
     close $ntr
     exit 0
}

# Procedimiento para imprimir en pantalla el nombre del nodo
# y el tiempo en el que ocurre un evento
Agent/Protocolo2Sender instproc registro_evento {time} {
     $self instvar node_
     puts "-----------------------------------"
     puts "NODO:[$node_ id], TIME: $time ms"
}

# Procedimiento para imprimir el nombre del nodo y el tiempo en el
# que ocurre un evento
Agent/Protocolo2Receiver instproc registro_evento {time} {
     $self instvar node_
     puts "-----------------------------------"
     puts "NODO:[$node_ id], TIME: $time ms"
}

# Creación de los nodos emisor(n0) y receptor(n1)
set n0 [$ns node]
set n1 [$ns node]

# Configuración del color de los nodos
$n0 color blue
$n1 color red

# Creación de los enlaces entre los nodos
$ns simplex-link $n0 $n1 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail
$ns simplex-link $n1 $n0 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail

# Creación del agente para el nodo emisor
set A [new Agent/Protocolo2Sender]
$A set frameSize_ $variable(frameSize_)

# Creación del agente para el nodo receptor
set B [new Agent/Protocolo2Receiver]

# Asignación de los agentes a los nodos
$ns attach-agent $n0 $A
$ns attach-agent $n1 $B

# Conexión entre los agentes
$ns connect $A $B
# Se define el color a los flujos de datos
$A set fid_ 1
$B set fid_ 2
```

```
# Inicio de la transmisión
$ns at $variable(iniciar) "$A iniciar_transmision"

# Fin de la simulación
$ns at $variable(detener) "finish"

$ns run
```

Script Protocolo2 Configuración del escenario para simular el protocolo "*simplex* de parada y espera".

Cabe indicar que aunque la transmisión es unidireccional la comunicación es en ambos sentidos, por tanto en el *script* de simulación es necesario configurar dos enlaces *simplex-link* como se indica en la Figura 3.19.

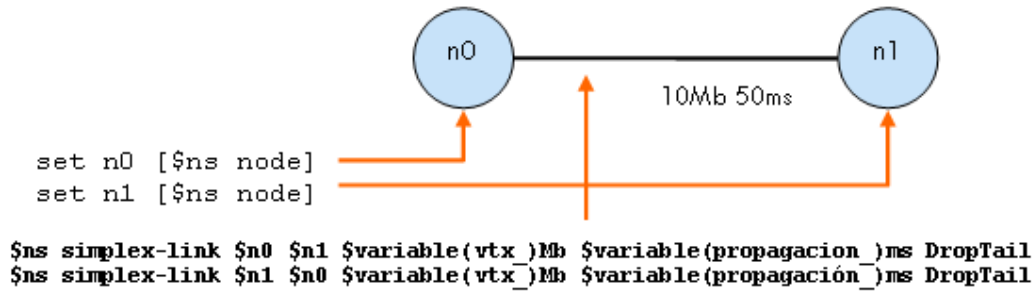


Figura 3.19 Configuración de dos enlaces unidirecionales en el script OTcl.

1.7.2.3. Resultados de la simulación

Los resultados de la simulación se los obtiene en los archivos out2.tr y out2.nam, además se puede observar los mensajes que se imprimen en pantalla.

1.7.2.3.1. Impresión en pantalla

A continuación, en la Figura 3.20 se presentan algunos de los resultados obtenidos de la simulación del protocolo "*simplex* de parada y espera".

Debido a que se ha configurado una velocidad de transmisión de 10 Mbps y un tamaño de trama de 1000 bytes se obtiene un tiempo de transmisión de 0.8 ms, el tiempo de propagación del canal es de 50ms, de manera que el tiempo total que se tarda una trama de datos en llegar al nodo receptor es de 50.8ms, mientras que la trama de confirmación cuyo tamaño es 10 bytes se tarda 50.08ms.

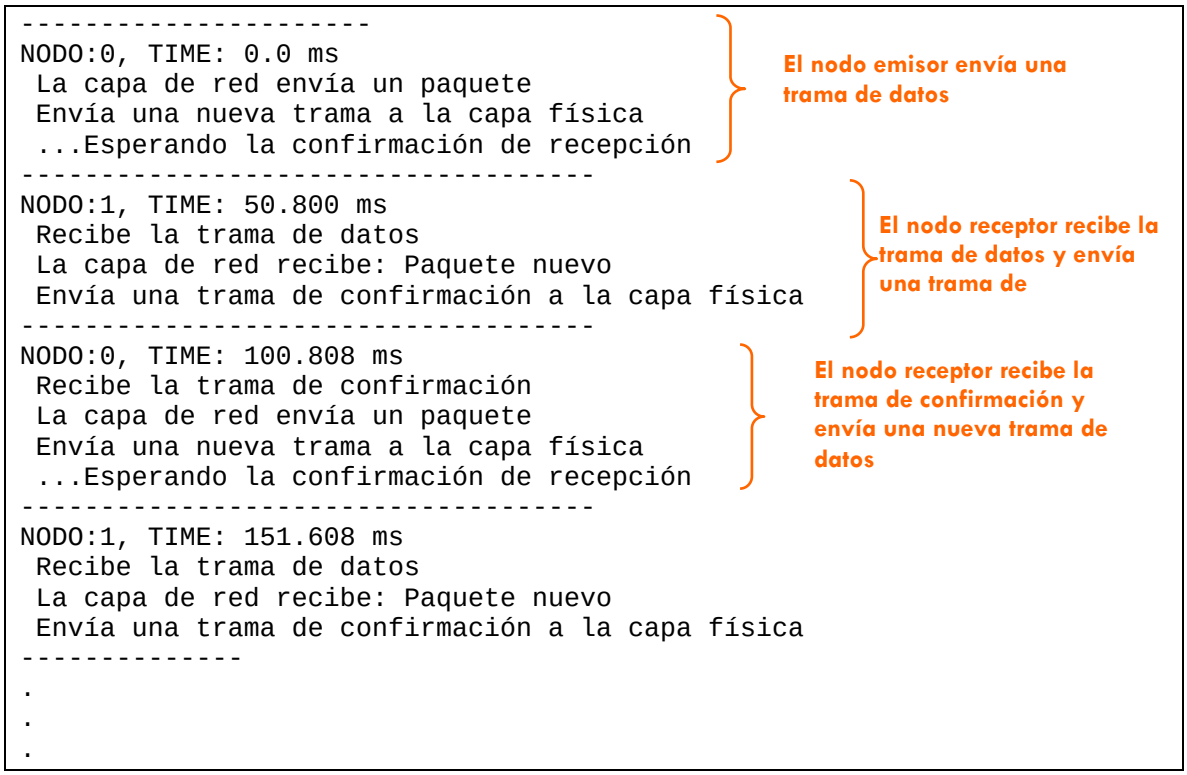


Figura 3.20 Resultados impresos en pantalla obtenidos de la simulación del protocolo "*simplex* de parada y espera".

1.7.2.3.2. Archivo out2.tr

En la Figura 3.21 se presenta parte del contenido del archivo out2.tr, en el cual se encuentran registrados los eventos que ocurrieron durante la simulación.

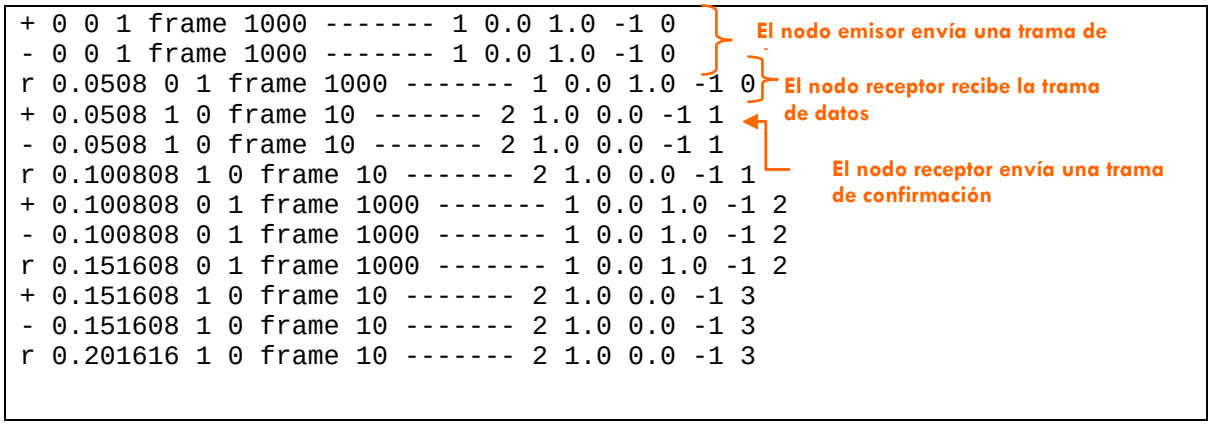


Figura 3.21 Resultados registrados en el archivo out2.tr.

El formato de presentación de los eventos registrados en el archivo out2.tr son similares a los presentados en el archivo out1.tr por lo tanto no se pondrá mucha énfasis en la explicación.

En los eventos registrados en este archivo se puede observar el tamaño diferente de la trama de datos y de las trama de confirmación.

```
+ 0.0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
- 0.058 1 0 frame 10 ------- 2 1.0 0.0 -1 1
```

Se puede observar cuando un nodo ha recibido una trama mediante el primer término en la línea que indica el se registro de dicho evento (*r*).

```
r 0.158 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
```

1.7.2.3.3. Archivo out2.nam

A continuación, en la Figura 3.22 y Figura 3.23 se indican la visualización de la transmisión de tramas que se produce al ejecutarse el archivo out2.nam obtenido de la simulación del protocolo "*simplex* de parada y espera".

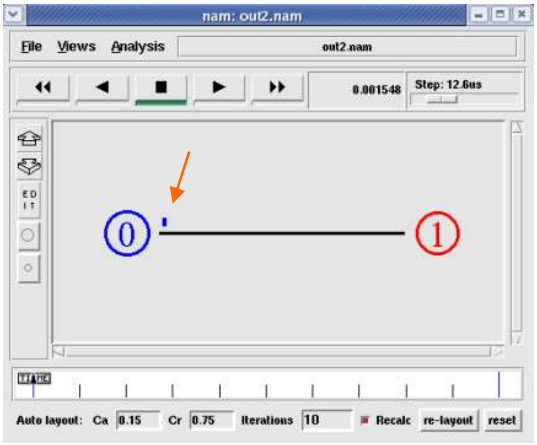


Figura 3.22 Envió de la trama de datos del nodo emisor hacia el nodo receptor.

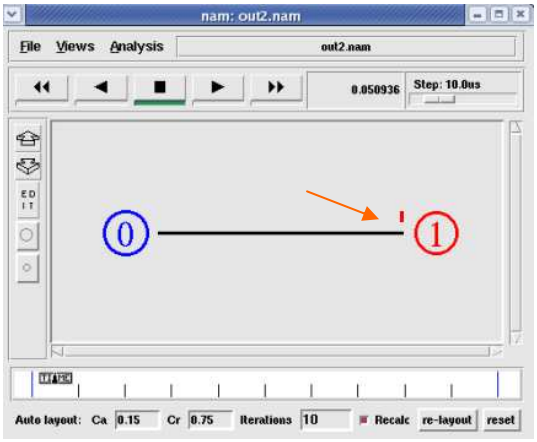


Figura 3.23 Envió de la trama de confirmación del nodo receptor hacia el nodo emisor.

De los resultados obtenidos se puede concluir que el protocolo cumple con las consideraciones que se hicieron para su implementación, específicamente con el

control de flujo; de esta manera se puede observar que una vez que el receptor recibe un trama éste envía una trama de confirmación al emisor (por ejemplo al tiempo 50.8ms el receptor recibe una trama de datos y envía una trama de confirmación), una vez que el emisor recibe la trama de confirmación puede enviar una nueva trama de datos (por ejemplo al tiempo 100.808ms el emisor recibe la trama de confirmación y envía una nueva trama).

1.7.3. PROTOCOLO SIMPLEX PARA UN CANAL CON RUIDO

1.7.3.1. Diseño e implementación en C++

Para solventar el problema mencionado en la descripción del protocolo "*simplex* para un canal con ruido", es necesario implementar un agente receptor que sea capaz de distinguir entre tramas que está recibiendo por primera vez y una retransmisión. La forma evidente de lograr esto es hacer que el emisor ponga un número de secuencia en cada trama que envía, de tal manera que el receptor pueda examinar el número de secuencia de la trama que llega para determinar si es una trama nueva o una trama duplicada que debe descartarse.

Para este caso un número de secuencia de un bit (0 ó 1) será suficiente, ya que en cada instante el receptor espera un número de secuencia en particular. Cualquier trama de entrada que contenga un número de secuencia equivocado se rechaza como duplicado. Cuando llega una trama que contiene el número de secuencia correcto, se acepta, se obtiene el paquete y se lo pasa a la capa de red, y el número de secuencia esperado cambia (si el valor es 0 se cambia a 1 y si el valor es 1 se cambia a 0).

En este protocolo se hace uso de un temporizador, el cual arranca luego que el emisor transmite una trama, si éste ya se estaba ejecutando se reestablece para conceder otro intervalo completo de temporización. Solo cuando ha transcurrido el intervalo completo de temporización (ya sea porque se ha perdido la trama de datos o la confirmación de recepción) el emisor puede suponer con seguridad que se ha perdido la trama y debe enviar su duplicado.

1.7.3.1.1. Temporizador

Los temporizadores son implementados mediante clases derivadas de la clase abstracta *TimerHandler* .

Clase TimerHandler

La definición de esta clase se la puede encontrar en el archivo *timer-handler.h,* para su ubicación ver el Anexo B.

A continuación se describen las principales variables y funciones de esta clase **[2]**.

Variables

- **status_:** Variable que almacena el estado del temporizador; puede tomar los valores definidos en el *enum TimerStatus* y son:

  TIMER_IDLE : timer expirado

  TIMER_PENDING : timer pendiente

  TIMER_HANDLING : timer que está siendo manejado

Funciones

- **sched():** Función que programa un temporizador para que expire luego de haber transcurrido el tiempo en segundos que es indicado en su argumento. La programación de un temporizador consiste en que un nodo envía un evento (objeto de la clase *Event*) a si mismo.
- **resched():** Función que reprograma la expiración de un temporizador que se encuentra en estado *TIMER_PENDING*.
- **cancel():** Función que cancela un temporizador que aún no ha expirado.
- **status():** Función que retorna el estado en el que se encuentra el temporizador.
- **expire():** Función virtual pura que debe ser necesariamente implementada por la clase derivada, dentro de ésta se implementará la funcionalidad que se debe realizar cuando un temporizador expire **[14]**.

Clase Protocolo3Timer

Clase implementada para representar a un temporizador, su definición se la presenta en el Código Protocolo 3.1.

```
class Protocolo3SenderAgent;
// Se crea una clase que hereda de TimerHandler en donde se
// redefinirán las funciones para el temporizador
class Protocolo3Timer : public TimerHandler{
public:
     Protocolo3Timer(Protocolo3SenderAgent* a) : TimerHandler()
{a_=a;};
 protected:
// Se invocará automáticamente cuando expire el temporizador
     virtual void expire(Event* e);
// Se crea un objeto de tipo Protocolo3SenderAgent
     Protocolo3SenderAgent* a_;
};
```

Código Protocolo 3.1 Definición de la clase *Protocolo3Timer***.**

Variables

- **a_:** Variable de tipo *Protocolo3SenderAgent* mediante la cual se puede hacer llamadas a funciones de esa clase.

Funciones

- **Protocolo3Timer():** Constructor en el que se reserva espacio de memoria que será asignado al puntero *a_* (ver el Código Protocolo 3.1).
- **expire():** Función redefinida de la clase *TimerHandler*; la función será invocada cuando el temporizador expire, en donde con el objeto *a_* se invoca a la función *retransmisión()* (que será explicada en la implementación de la clase *Protocolo3SenderAgent*). En el Código Protocolo 3. 2) se presenta su implementación.

```
void Protocolo3Timer::expire(Event*)
{
     // En caso de que el nodo emisor reciba un evento que
     // indique que el temporizador ha expirado, entonces
     // la función expire sera invocada automáticamente
     a_->registro_evento();
     printf(" !!! Temporizador expirado\n");

     // Se  invoca a la función retransmisión de
     // Protocolo3SenderAgent para que retransmita la trama perdida
     a_->retransmision();

}
```

Código Protocolo 3.2 Función *expire()* de la clase *Protocolo3Timer***.**

1.7.3.1.2. Agentes

Clase Protocolo3SenderAgent

El objetivo de esta clase es transmitir una trama de datos y arrancar un temporizador, luego de esto esperar que ocurra uno de los siguientes eventos:

- Llegada de una trama de confirmación de recepción.
- Expiración del temporizador.

El Código Protocolo 3.3 presenta la definición de la clase *Protocolo3SenderAgent.*

```
#define MAX_SEQ 1       // Maximo número que tendra la secuencia
typedef int seq_nr;     // Número de secuencia o de confirmación

class Protocolo3SenderAgent: public Agent{
private:
// Variables del protocolo

     frame  *s_;                    // Variable que representa a la
                                    // trama que será enviada
     frame  *r_;                    // Variable que representa a la
                                    // trama que será recibida
     paquete buffer;                // Paquete que se obtiene de la
                                    // capa de red
     Packet *p_s_;                  // Variable con la que se podrá
                                    // acceder a frame(hdr_frame)
     int frameSize_;                // Tamaño de la trama
     double timer_;                 // Variable que representa el
                                    // intervalo de tiempo que espera
                                    // antes que el temporizador expire
     seq_nr next_frame_to_send;          // Variable que representa
                                         // el número de secuencia de
                                         // la siguiente trama de salida
     Protocolo3Timer protocolo_timer_;   // Variable que representa
                                         // al temporizador
public:
// Constructor
     Protocolo3SenderAgent();
// Destructor
     ~Protocolo3SenderAgent();
// Funciones del protocolo
     void from_network_layer(paquete * msg);   // Obtiene un paquete
                                               // de la capa de red
     void to_physical_layer(Packet * psend);   // Envía una trama
                                               // hacia la capa física
     void initialize();                        // Empieza con la
                                               // transmisión de datos
// Funciones de la clase Agent que se redefinen
     int command (int argc, const char*const* argv);
     void recv(Packet* p_r_, Handler*);
// Funciones para el temporizador
     void start_timer();                       // Inicia el temporizador
     void stop_timer();                        // Detiene el temporizador
```

```
// Funciones auxiliares
     void crear_paquete(paquete *msg);   // Genera un paquete en
                                         // la capa de red
     void registro_evento();             // Función que invoca a un
                                         // procedimiento en OTcl para
                                         // imprimir el nombre del nodo
                                         // al cual está asignado este
                                         // agente
     void inc(int& k);                   // Incrementa circularmente
                                         // el el valor de k
     void retransmision();               // Retransmite una trama cuando
                                         // expira el temporizador.
};
```

Código Protocolo 3.3 Definición de la clase *Protocolo3SenderAgent*.

Variables

- **next_frame_to_send_:** Variable con la que el emisor recuerda el número de secuencia de la siguiente trama a enviar.
- **timer_:** Variable que representa el intervalo de tiempo en el que el temporizador expirará. Dicho intervalo debe seleccionarse de modo que haya suficiente tiempo para que la trama llegue al receptor, sea procesada y retorne la confirmación de recepción al emisor.
- **protocolo_timer_:** Variable que representa al temporizador y con la cual se accederá a funciones del mismo.

Funciones

- **Protocolo3SenderAgent():** Invoca al constructor de *Protocolo3Timer* enviando como argumento un puntero a si mismo (*this*), para que de esta manera el objeto *protocolo_timer_* pueda acceder a las funciones de la clase *Protocolo3SenderAgent*. Adicionalmente se inicializan sus variables como se muestra en el Código Protocolo 3.4.

```
Protocolo3SenderAgent::Protocolo3SenderAgent() :
Agent(PT_FRAME),protocolo_timer_(this)
{

     // Variables que serán configuradas desde el script OTcl
     bind("frameSize_",&frameSize_);
     bind("timer_",&timer_);

}
```

Código Protocolo 3.4 Implementación del constructor de la clase *Protocolo3SenderAgent.*

- **start_timer():** Función que programa el intervalo de tiempo (*timer_*) en el que expirará el temporizador. Para esto se hace uso de la función *resched()* de la clase *TimerHandler* que es invocada a través del objeto *protocolo_timer_.* Su implementación se la presenta en el Código Protocolo 3.5.

```
void Protocolo3SenderAgent::start_timer()
{
     printf(" Iniciando el temporizador\n");

     // Se reprograma el temporizador
     protocolo_timer_.resched(timer_);

}
```

Código Protocolo 3.5 Implementación de la función *star_timer()* de la clase *Protocolo3SenderAgent.*

- **stop_timer():** Función que cancela la llegada de un temporizador que ha sido programado; lo hace a través de la función *cancel()* de la clase *TimerHandler* como se indica en el Código Protocolo 3.6.

```
void Protocolo3SenderAgent::stop_timer()
{
     printf(" Detiene el temporizador\n");

     // Se cancela el temporizador
     protocolo_timer_.cancel();

}
```

Código Protocolo 3.6 Implementación de la función *stop_timer()* de la clase *Protocolo3SenderAgent***.**

- **initialize():** Función que empieza con la transmisión; en ella se crea únicamente la primera trama a la cual se le asigna un número de secuencia, posteriormente se crea una copia de la trama para enviar a la capa física ya que la original se la debe mantener en caso que sea necesario una retransmisión, finalmente se programa un temporizador en cuyo intervalo de tiempo el módulo emisor deberá recibir una confirmación de recepción de la trama enviada. En Código Protocolo 3.7 se presenta su implementación.

```
void  Protocolo3SenderAgent::initialize()
{
     // Reserva un espacio de memoria para el
     // puntero p_s_ y configura algunos de los
     // campos para la estructura hdr_cmd
     p_s_= allocpkt();

     // Función que imprimirá en pantalla el nombre
     // del nodo y el tiempo en el que llega un evento
     registro_evento();

     // Se inicia el número de secuencia de salida
     next_frame_to_send=0;

     // Con el Packet s_ se accede a frame(hdr_frame)
     s_ = frame::access(p_s_);

     // Se obtiene el primer paquete de la capa de red
     from_network_layer(&buffer);

     // Se coloca el paquete en el campo info de la trama
     s_->info_var=buffer;

     // Se inserta el número de secuencia en la trama
     s_->seq_var=next_frame_to_send;

     // Con el Packet p_s_ se accede a hdr_cmn
     hdr_cmn *ch = hdr_cmn::access(p_s_);

     // Se asigna el tamaño de la trama
     ch->size()=frameSize_;

     // Se crea un objeto Packet que será enviado
     Packet *copia;

     // Se crea una copia del p_s_ para enviar
     copia=p_s_->copy();

     // Se envía una copia a la capa física
     to_physical_layer(copia);

     // Se inicia el temporizador
     start_timer();

}
```

Código Protocolo 3.7 Implementación de la función *initialice()* de la clase *Protocolo3SenderAgent*.

- **inc():** Función que incrementa en uno el valor de la variable obtenida del argumento hasta que llegue a cierto límite (MAX_SEQ); al llegar al límite, el valor de la variable retorna a cero y nuevamente se comienza a incrementar su valor. En el Código Protocolo 3.8 se muestra su implementación.

```
void Protocolo3SenderAgent::inc(int& k)
{
     if(k<MAX_SEQ)
     {     k=k+1;
     }
     else
     {     k=0;
     }
}
```

Código Protocolo 3.8 Implementación de la función *inc()* del clase *Protocolo3SenderAgent*.

- **retransmision():** Función que será invocada en la función *expire()* de la clase *Protocolo3Timer*. Aquí se crea un nuevo objeto de la clase *Packet*, el cual será una réplica del puntero *p_s_* anteriormente enviado; la copia se la realiza mediante la función *copy()* de la clase *Packet*. En el Código Protocolo 3.9 se muestra la implementación de la función.

```
void Protocolo3SenderAgent::retransmision()
{

     // Se crea un objeto Packet que será enviada
     Packet *copia;

     // Se copia los valores de los campos de
     // la trama que anteriormente fue enviada
     copia=p_s_->copy();

     // Se envía la trama a la capa física
     to_physical_layer(copia);

     // Se inicia el temporizador
       start_timer();
}
```

Código Protocolo 3.9 Implementación de la función *retransmisión()* de la clase *Protocolo3SenderAgent*.

- **recv():** Función que recibe la trama de confirmación en donde si el número de confirmación de la trama recibida *(r_->ack_var)* es igual que el número de secuencia de la última trama que envió (*next_frame_to_send)*, entonces se detiene el temporizador (*stop_timer()*), se obtiene un nuevo paquete de la capa de red y se incrementa el número de secuencia con la función *inc()*; posteriormente, se crea una nueva trama para enviar a la capa física. En el Código Protocolo 3.10 se presenta su implementación.

```
void Protocolo3SenderAgent::recv(Packet* p_r_, Handler*)
{
     // Imprime el nombre del nodo  y el tiempo
     registro_evento();
     // Con el Packet p_r_ e accede a frame
     r_ = frame::access(p_r_);
     printf("Recibe el ack de la trama: %d \n",r_->ack_var);

     // Si la trama que llega tiene el número acuse de recibo
     // esperado, entonces se detiene el temporizador y
     // se obtiene un nuevo paquete de la capa de red

     if(r_->ack_var==next_frame_to_send)
     {
           // Se detiene al temporizador
           stop_timer();

           // Se obtiene un paquete de la capa de red
           from_network_layer(&buffer);

           // Se cambia el número de secuencia de la siguiente trama
           // que se enviará
           inc(next_frame_to_send);
     }

     // Borra la trama recibida
     delete p_r_;

     // Con el Packet p_s_ se accede
     // a frame(hdr_frame)
     s_ = frame::access(p_s_);

     // Se coloca el paquete en
     // el campo info de la trama
     s_->info_var=buffer;

     // Se inserta el número de secuencia en la trama
     s_->seq_var=next_frame_to_send;

     // Con el Packet copia se accede a hdr_cmn
     hdr_cmn *ch = hdr_cmn::access(p_s_);

     // Se asigna el tamaño de la trama
     ch->size()=frameSize_;

     // Se crea un objeto Packet que será enviado
     Packet *copia;

     // Se crea una copia del p_s_ para enviar
     copia=p_s_->copy();

     // Se envía una copia a la capa física
     to_physical_layer(copia);

     // Se inicia el temporizador
     start_timer();

}
```

Código Protocolo 3.10 Implementación de la función *recv()* de la clase *Protocolo3SenderAgent.*

Clase Protocolo3ReceiverAgent

El objetivo de esta clase es recibir una trama de datos y generar una trama de control para enviar como confirmación la recepción.

En el Código Protocolo 3.11 se presenta la definición de la clase *Protocolo3ReceiverAgent.*

```
class Protocolo3ReceiverAgent: public Agent{

private:
// Variables del protocolo
     frame *s_;              // Variable que representa a la
                             // trama que será enviada
     frame *r_;              // Variable que representa a la
                             // trama que será recibida
     Packet *p_s_;           // Variable con la que se podrá
                             // acceder a frame(hdr_frame)
     int frameSize_;         // Tamaño de la trama
     seq_nr frame_expected;  // Variable que representa
                             // al número de secuencia de la
                             // trama que se espera recibir
public:
// Constructor
     Protocolo3ReceiverAgent();
// Destructor
     ~Protocolo3ReceiverAgent();
// Funciones del protocolo

     void to_network_layer(paquete *msg);     // Envía un paquete a
                                              // la capa de red
     void to_physical_layer(Packet * psend);  // Envía una trama
                                              // hacia la capa física

// Funciones de la clase Agent que se redefinen
     int command (int argc, const char*const* argv);
     void recv(Packet* p_r_, Handler*);
//Funciones auxiliares
     void registro_evento(); // Función que invoca un
                             // procedimiento OTcl para
                             // imprimir el nombre del nodo
                             // y el tiempo en que ocurre un
                             // evento en éste
     void inc(int& k);       // Incrementa circularmente
                             // el valor de la trama que se está
                             // esperando recibir
};
```

Código Protocolo 3.11 Definición de la clase *Protocolo3ReceiverAgent.*

Variables

- **s_:** Variable que representa a la trama de control generada para la confirmación de recepción, en ésta sólo se configurará el campo *ack_var*

de la trama, con el cual se le indica al emisor que trama llegó correctamente.

- **frame_expected:** Variable que representa el número de secuencia de la trama que espera recibir.

Funciones

- **recv():** Función que es invocada al llegar una trama válida al receptor en donde se verificará si el número de secuencia de la trama que recibe (*r_—>seq_var*) corresponde a la trama que esperaba recibir (*frame_expected*); si esto se cumple se obtiene el paquete, se lo envía a la capa de red y se cambia el valor de *frame_expected*; posteriormente, se envía la trama de confirmación. Su implementación se encuentra en el Código Protocolo 3.12.

```
void Protocolo3ReceiverAgent::recv(Packet* p_r_, Handler*)
{
     // Función que imprimirá en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();
     // Con p_r_ se accede a frame
     r_ = frame::access(p_r_);
     printf(" Recibe la trama con seq: %d\n",r_->seq_var);

     // Se verifica si el número de secuencia de la
     // trama recibida es la misma que la esperada
     if(r_->seq_var==frame_expected)
     {
           // Se envía el paquete a la capa de red
           to_network_layer(&r_->info_var);
           // Se incrementa el valor de frame_expected
           inc(frame_expected);
     }
     // Se borra la trama recibida
     delete p_r_;
     p_s_=allocpkt();
     // Con el Packet p_s_ se accede a
     // frame (hdr_frame)
     s_ = frame::access(p_s_);
     // Se inserta el número de confirmación (0 o 1) de la trama
     s_->ack_var=1-frame_expected;
     // Se accede a la cabecera hdr_cmn
     hdr_cmn *ch = hdr_cmn::access(p_s_);
     // Se asigna el tamaño de la trama
    ch->size()=frameSize_;
     // Se envía la trama a la capa física
    to_physical_layer(p_s_);
}
```

Código Protocolo 3.12 Implementación de la función *recv()* de la clase *Protocolo3ReceiverAgent*.

1.7.3.1.3. Canal con Ruido

Para la simulación de un canal con ruido es necesario implementar un modelo de error utilizando las distribuciones de probabilidad existentes en NS-2 y la funcionalidad de la clase *ErrorModel* implementada en C++ y OTcl.

Distribuciones

La probabilidad de error ó pérdida de cualquier unidad de datos (para este caso de la trama), dependerá de la distribución de probabilidad utilizada.

En NS-2 las principales distribuciones de probabilidad utilizadas son **[2]**:

- *Pareto Distribution*
- *Constant*
- *Uniform Distribution*
- *Exponential Distribution*
- *Hyperexponential Distribution*

Para mayor detalle se puede revisar la referencia **[5]**.

Clase ErrorModel

Clase derivada de *Connector*, proporciona funcionalidad con la que permite modelar un canal de comunicaciones con ruido, mediante la introducción de la probabilidad de pérdida de información o introducción de errores en objetos de la clase *Packet* (objetos que son intercambiados entre los agentes).

Las principales variables de la clase son **[2]**:

- **unit_:** Representa a la unidad de información que puede contener un error o que puede perderse; los valores que puede tomar son: *packet*, *time*, *bit*.
- **ranvar_:** Representa a la variable aleatoria para la generación de error.

Si no se configuran las variables antes mencionadas, tomarán valores por defecto que son: *unit_* será asignada con *packet_* y *ranvar_* será asignado con una distribución de probabilidad uniforme.

- **rate_:** Representa la tasa con la que se produce un error ó pérdida en la transmisión de las unidades de datos.

1.7.3.2. Configuración del escenario de simulación

```
#===================================================================#
# Parámetros que ingresará el usuario                               #
#===================================================================#
set variable(propagacion_) 50       ;# Tiempo de propagación en el
                                    ;# canal en ms
set variable(vtx_)         10       ;# Velocidad de transmisión en Mbps
set variable(frameSize_)   1000     ;# Tamaño de la trama en bytes
set variable(nam)          out3.nam ;# Archivo de trazas .nam
set variable(tr)           out3.tr  ;# Archivo de trazas .tr
set variable(timer_)       0.120    ;# Tiempo para que el temporizador
                                    ;# expire
set variable(loss_)        0.01     ;# Tasa de pérdida de tramas
set variable(iniciar)      0.0      ;# Inicio de la transmisión
set variable(detener)      1.0      ;# Fin de la simulación
#===================================================================#
# Se crea un objeto de la clase Simulator
set ns [new Simulator]

# Diferentes colores para los flujos de datos
$ns color 1 Blue
$ns color 2 Red

# Creación del archivo de trazas .nam
set nf [open $variable(nam) w]
$ns namtrace-all $nf
# Creación del archivo de trazas .tr
set ntr [open $variable(tr) w]
$ns trace-all $ntr

# Definición del procedimiento finish que es invocado para
# finalizar la simulación
proc finish {} {
     global ns nf ntr
     $ns flush-trace
     close $nf
     close $ntr
     exit 0
}
# Procedimiento para imprimir el nombre del nodo y el tiempo en el
# que ocurre un evento
Agent/Protocolo3Sender instproc registro_evento {time} {
.    $self instvar node_
     puts "-----------------------------------"
     puts "NODO:[$node_ id], TIME: $time ms"
}
# Procedimiento para imprimir el nombre del nodo y el tiempo en el
# que ocurre un evento
Agent/Protocolo3Receiver instproc registro_evento {time} {
     $self instvar node_
     puts "-----------------------------------"
     puts "NODO:[$node_ id], TIME: $time ms"
}
```

```
# Creación de los nodos emisor(n0) y receptor(n1)
set n0 [$ns node]
set n1 [$ns node]
# Configuración del color a los nodos
$n0 color blue
$n1 color red

# Creación de los canales que enlazan los nodos
$ns simplex-link $n0 $n1 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail
$ns simplex-link $n1 $n0 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail

# Creación del agente emisor y configuración de sus parámetros
set A [new Agent/Protocolo3Sender]
$A set timer_ $variable(timer_)
$A set frameSize_ $variable(frameSize_)

# Creación del agente receptor y configuración de sus parámetros
set B [new Agent/Protocolo3Receiver]
$B set frameSize_ $variable(frameSize_)

# Asignación de los agentes a los nodos
$ns attach-agent $n0 $A
$ns attach-agent $n1 $B

# Conexión entre los agentes
$ns connect $A $B
# Se define el color a los flujos de datos
$A set fid_ 1
$B set fid_ 2
#-------------------------------------------------------#
#               CANAL CON RUIDO                         #
#-------------------------------------------------------#
# Procedimiento que crea un modelo de pérdida en el canal
proc canal_perdida {tasa_perdida nodo0 nodo1} {
     global ns
     set mod_perdida [new ErrorModel]
     # Tasa de pérdida en el canal
     $mod_perdida set rate_ $tasa_perdida ;
     # Se configura la variable aleatoria para
     # la generación de pérdida en el canal
     $mod_perdida ranvar [new RandomVariable/Uniform]
     # El agente Null recibirá los paquetes perdidos
     # en el canal
     $mod_perdida drop-target [new Agent/Null]
     # Enlace al cual se aplicará el modelo de pérdida
     $ns lossmodel $mod_perdida $nodo0 $nodo1
}
# Asignación de un modelo de pérdida para cada conexión
canal_perdida $variable(loss_) $n1 $n0
canal_perdida $variable(loss_) $n0 $n1

# Inicio de la transmisión
$ns at $variable(iniciar) "$A iniciar_transmision "
# Fin de la simulación
$ns at $variable(detener) "finish"
$ns run
```

Script Protocolo3 Configuración del escenario para simular el protocolo "*simplex* para un canal con ruido".

Como se mencionó anteriormente, en este protocolo se requiere modelar un canal con ruido (que se ve reflejado en la pérdida de unidades de datos), lo que se lo ha realizado desde el *script* de simulación.

Antes de indicar el proceso de configuración, es necesario describir algunos procedimientos que son usados para este objetivo.

- **drop-target:** Procedimiento de la clase *ErrorModule*[1] en el que se añade un agente nulo (*Agent/Null*) el cual recibirá los paquetes que no llegan a su destino.
- **lossmodel:** Procedimiento de la clase *Simulator* que permite introducir el modelo de error en el canal de comunicación (enlace) que se desee. Toma los siguientes argumentos: modelo de error (*lossobj*), el nodo origen del enlace (*from*) y nodo destino del enlace (*to*).

```
Simulator instproc lossmodel {lossobj from to} {
     set link [$self link $from $to]
     $link errormodule $lossobj
}
```

- **Agente Nulo:** Este agente es utilizado como un sumidero para las unidades de datos (objetos de tipo *Packet*) que se han perdido en el canal de comunicación o como destino de las unidades de datos que no son tomadas en cuenta **[2]**. En la Figura 3.24 se presenta la ubicación del Agente Nulo en el enlace entre dos nodos. La implementación de la clase se encuentra en el archivo ns-agent.tcl para su ubicación ver el Anexo B.

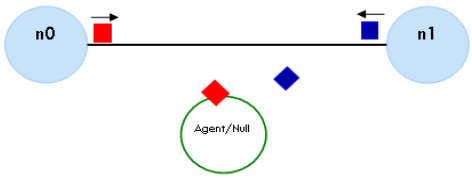


Figura 3.24 Ubicación del agente nulo en el enlace entre dos nodos.

La configuración del modelo de pérdida se basa en el siguiente procedimiento:

1. Se crea una variable que represente al modelo de pérdida al cual se le configura la tasa de pérdida de paquetes y la distribución de probabilidad que va a ser usada para ese fin.

[1] Clase derivada de la clase *Connector* que utiliza objetos de la clase *Classifier* a los cuales se añade los modelos de error que se configure.

```
# Se crea el modelo pérdida
set mod_perdida [new ErrorModel]

# Se configura la tasa de pérdida de paquetes
$mod_perdida set rate_ $tasa_perdida;

# Se configura la districisión de probabilidad con la que se
# generará el error
$mod_perdida ranvar [new RandomVariable/Uniform]
```

2. Se invoca al procedimiento *drop-target* enviando como argumento al agente nulo que procesará los paquetes que se pierdan en el canal de comunicación.

```
$mod_perdida drop-target [new Agent/Null]
```

3. Se asigna el modelo de pérdida en el enlace entre los dos nodos.

```
$ns lossmodel $mod_perdida $nodo0 $nodo1
```

1.7.3.3. Resultados de la simulación

1.7.3.3.1. Impresión en pantalla

A continuación, en la Figura 3.25 se indican los resultados que se imprimen en pantalla al realizar la simulación del protocolo "*simplex* para un canal con ruido".

```
-----------------------------------
NODO:0, TIME: 0.000 ms
 La capa de red envía un paquete
 Envía la trama con seq : 0
 Iniciando el temporizador
-----------------------------------
NODO:0, TIME: 120.000 ms
 !!! Temporizador expirado
 Envía la trama con seq : 0
 Iniciando el temporizador
-----------------------------------
NODO:1, TIME: 170.800 ms
 Recibe la trama con seq: 0
 La capa de red recibe: Nuevo Paquete
 Envía el ack de la trama: 0
-----------------------------------
NODO:0, TIME: 221.600 ms
 Recibe el ack de la trama: 0
 Detiene el temporizador
 La capa de red envía un paquete
 Envía la trama con seq : 1
 Iniciando el temporizador
.
.
```

Se envía una trama con secuencia 0 desde el nodo emisor

Expira el temporizador de la trama con secuencia 0

El nodo receptor recibe la trama y envía una trama de confirmación

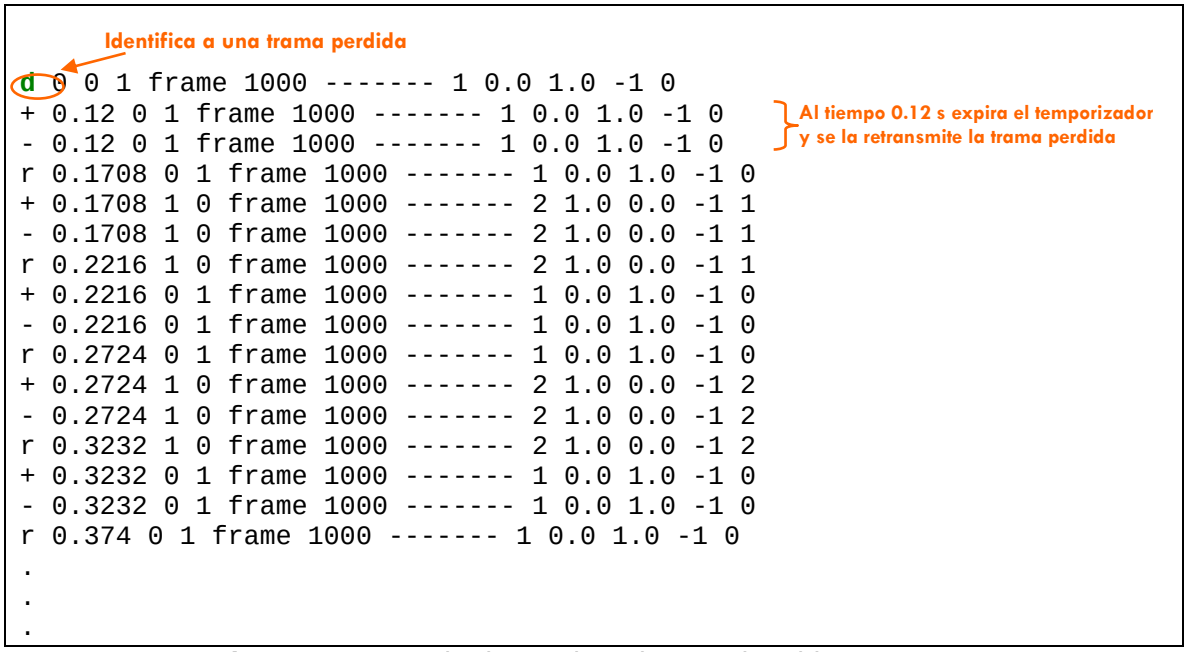
Figura 3.25 Resultados impresos en pantalla obtenidos de la simulación del protocolo "*simplex* para un canal con ruido".

En los resultados presentados en la Figura 3.25 se puede apreciar:

- El nodo emisor al tiempo 0ms envía una trama con secuencia 0 e inicia el temporizador; al tiempo 120.0ms expira el temporizador debido a que la trama se perdió, por tanto el emisor procede a reenviar la trama.
- Al tiempo 170.0ms el nodo receptor recibe la trama esperada, por tanto obtiene el paquete del campo de datos y lo pasa a la capa de red, posteriormente, envía una trama de confirmación de la trama recibida. Al tiempo 221.6ms el nodo emisor recibe la trama de confirmación de la última trama enviada, por tanto detiene el temporizador y obtiene un nuevo paquete de la capa de red.

1.7.3.3.2. Archivo out3.tr

En la Figura 3.26 se presentan los resultados que se registran en el archivo out3.tr.




```
d 0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
+ 0.12 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
- 0.12 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
r 0.1708 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
+ 0.1708 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
- 0.1708 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
r 0.2216 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
+ 0.2216 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
- 0.2216 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
r 0.2724 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
+ 0.2724 1 0 frame 1000 ------- 2 1.0 0.0 -1 2
- 0.2724 1 0 frame 1000 ------- 2 1.0 0.0 -1 2
r 0.3232 1 0 frame 1000 ------- 2 1.0 0.0 -1 2
+ 0.3232 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
- 0.3232 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
r 0.374 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
.
.
.
```

Figura 3.26 Resultados registrados en el archivo out3.tr.

En el archivo *out3.tr* se puede discriminar una trama perdida en el canal de comunicación mediante el primer término (d) de la línea que registra el evento.

```
d 0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
```

1.7.3.3.3. Archivo out3.nam

A continuación, en la Figura 3.27, Figura 3.28, Figura 3.29 se presentan los resultados que se visualizan con la ejecución del archivo out3.nam obtenido de la simulación del protocolo "*simplex* para un canal con ruido".

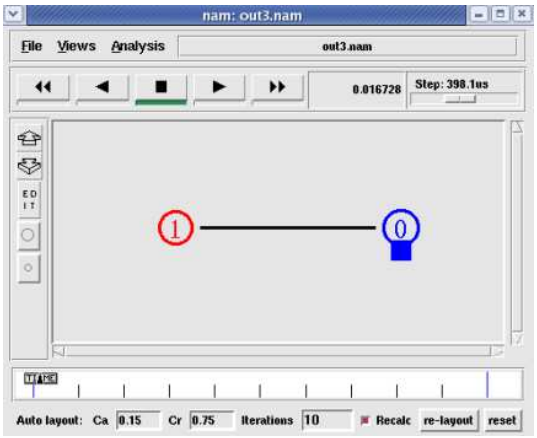


Figura 3.27 Pérdida de una trama en el canal de comunicaciones.

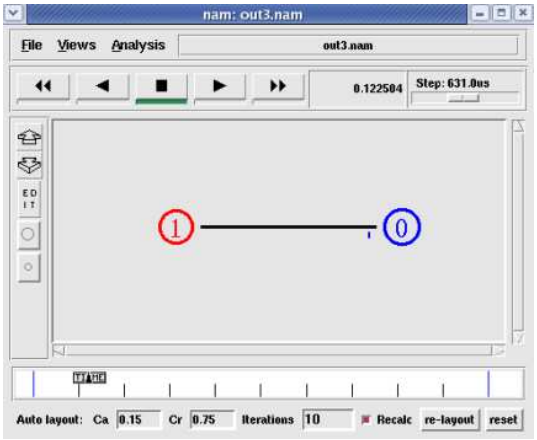


Figura 3.28 Envío de una trama de datos del nodo emisor al nodo receptor.

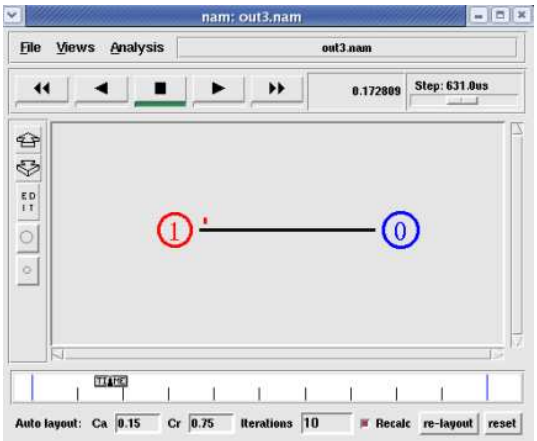


Figura 3.29 Envió de una trama de confirmación del nodo receptor al nodo emisor.

De los resultados obtenidos se puede concluir que el protocolo cumple con las consideraciones descritas para su implementación, es así que si la trama llega a su destino esta es aceptada y se devuelve una trama de confirmación pora ésta. En el caso que se pierda la trama, el temporizador para ésta expira y por tanto se realiza la retransmisión para dicha trama.

1.8. IMPLEMENTACIÓN DE PROTOCOLOS DE VENTANA CORREDIZA

1.8.1. PROTOCOLO DE VENTANA CORREDIZA DE UN BIT

1.8.1.1. Diseño e implementación en C++

Para la implementación se ha considerado los dos escenarios planteados en la descripción del protocolo de “ventana corrediza de un bit” (ver la Sección 2.2.1.1).

La máquina que arranca obtiene el primer paquete de su capade red, lo almacena en el búfer, lo inserta en el campo de datos de la trama que será enviada. Al llegar la trama al receptor, la capa de enlace de datos revisa el número de secuencia para saber si es un duplicado o no, igual que en el protocolo “*simplex* para un canal con ruido”. Si la trama es la esperada, se obtiene el paquete y se envía a la capa de red; además se cambia el valor de la trama a esperar.

El campo de confirmación de recepción de la trama recibida contiene el número de la última trama recibida sin error en la otra máquina. Si este número concuerda con el valor de la siguiente trama que está tratando de enviar el emisor (a su vez será el número de secuencia de esta trama); entonces éste sabrá que la última trama enviada ha llegado correctamente al receptor, por tanto, puede obtener el siguiente paquete de su capa de red para almacenarlo en el búfer. Si el valor del campo de confirmación no concuerda, deberá enviar una copia de la última trama enviada.

Debido a que la transmisión de datos es bidireccional, se ha implementado un solo agente que cumpla con el rol de emisor y receptor mediante la clase *Protocolo4Agent*.

Al igual que en el protocolo "*simplex* para un canal con ruido", para solucionar el problema que surge cuando existe pérdida de tramas, se han implementado temporizadores con la misma lógica y funcionalidad.

1.8.1.1.1. Agente

Clase Protocolo4Agent

A continuación se presenta la definición de la clase *Protocolo4Agent.*

```
#define MAX_SEQ 1      // Máximo número que tendrá la secuencia
typedef int seq_nr;    // Número de secuencia o de confirmación

class Protocolo4Agent: public Agent{
private:
//Variables del protocolo
     frame  *s_;       // Variable que representa a la
                       // trama que será enviada
     frame  *r_;       // Variable que representa a la
                       // trama que será recibida
     paquete buffer;   // Paquete que se obtiene  de la
                       // capa de red
     Packet *p_s_;     // Variables con la que se podrá
                       // acceder a frame(hdr_frame)
     int frameSize_;   // Tamaño de la trama
     double timer_;    // Variable que representa el
                       // intervalo de tiempo que espera
                       // antes que el temporizador expire
     seq_nr next_frame_to_send;    // Variable que representa
                                   // el número de secuencia de
                                   // la siguiente trama de salida
     seq_nr frame_expected;        // Variable que representa
                                   // al número de secuencia de
                                   // la trama que espera recibir
     Protocolo4Timer protocolo_timer_;   // Variable que representa
                                         // al temporizador
public:

// Constructor
     Protocolo4Agent();

// Destructor
     ~Protocolo4Agent();

// Funciones del protocolo
     void from_network_layer(paquete * msg);   // Obtiene un paquete
                                               // de la capa de red
     void to_network_layer(paquete *msg);      // Envía un paquete a
                                               // la capa de red
     void to_physical_layer(Packet * psend);   // Envia una trama
                                               // hacia la capa física
     void initialize();                        // Empieza con la
                                               // transmisión de datos
```

```
// Funciones de la clase Agent que se redefinen
     int command (int argc, const char*const* argv);
     void recv(Packet* p_r_, Handler*);

// Funciones para el temporizador
     void start_timer();      // Inicia el temporizador
     void stop_timer();       // Detiene el temporizador


// Funciones auxiliares
     void crear_paquete(paquete *msg);   //Genera un paquete en la capa
                                         // de red
     void registro_evento(); // Función que invoca un
                             // procedimiento OTcl para imprimir
                             // el nombre del nodo y el
                             // tiempo en que ocurre un evento
     void inc(int& k);       // Incrementa circularmente
                             // el valor del argumento
     void retransmision();   // Retransmite una trama cuando
                             // se expira el temporizador
};
```

Código Protocolo 4.1 Definición de la clase *Protocolo4Agent*.

Funciones

- **initialize():** Función que será invocada para iniciar con la transmisión de la primera trama. En ella se obtiene un paquete de la capa de red, se construye una trama asignándole el número de secuencia, el número de confirmación y se la envía a la capa física; adicionalmente, se inicia un temporizador para la espera de confirmación de la trama que se está enviando (ver el Código Protocolo 4.2).

```
void  Protocolo4Agent::initialize()
{
     // Reserva un espacio de memoria para el puntero p_s_ y
     // configura algunos de los campos para la estructura hdr_cmd
     p_s_= allocpkt();

     // Función que imprimirá en pantalla
     // el nombre del nodo y el tiempo
     registro_evento();

     // Se obtiene el primer paquete de la capa de red
     from_network_layer(&buffer);

     // Con el Packet p_s_ se accede a frame (hdr_frame)
     s_ = frame::access(p_s_);

     // Se inserta el número de secuencia en la trama
     s_->seq_var=next_frame_to_send;
     // Se inserta el número de confirmación de la trama
     s_->ack_var=1-frame_expected;
```

```
    // Se coloca el paquete en el campo info de la trama
    s_->info_var=buffer;

    // Con el Packet p_s_ se accede a hdr_cmn
    hdr_cmn *ch = hdr_cmn::access(p_s_);

    // Se asigna el tamaño de la trama
    ch->size()=frameSize_;

    // Se crea un objeto Packet que será enviado
    Packet *copia;

    // Se crea una copia de p_s_ para enviar
    copia=p_s_->copy();

    // Se envía una copia a la capa física
    to_physical_layer(copia);

    // Se inicia el temporizador
    start_timer();
}
```

Código Protocolo 4.2 Implementación de la función *initialize()* de la clase *Protocolo4Agent*.

- **recv():** Función en la que se implementa la funcionalidad de emisor y receptor. Debido a que se está implementando la técnica de superposición *(piggybacking*), en esta función se verifica tanto el campo de número de secuencia como el de confirmación de recepción para tomar una decisión.

  Al verificar si el número de secuencia de la trama recibida (*r_─>seq_var*) es el mismo que el que se esperaba recibir (*frame_expected*) se procede a obtener el paquete y enviarlo a la capa de red, se incrementa el valor de la variable *frame_expected* (de acuerdo al tamaño de la ventana), es decir si el valor de *frame_expected* era 0, se cambiará a 1, y si el valor era 1 se cambiará a 0.

  Posteriormente, se verifica si el número de confirmación (*r_->ack_var)* contenido en la trama recibida es el mismo que el número de secuencia de la última trama que fue enviada (*next_frame_to_send*); si ese es el caso, se obtiene un nuevo paquete de la capa de red y se incrementa el valor de la variable *next_frame_to_send* (de la misma forma que se indicó para *frame_expected*). Además, se borra el objeto de la clase *Packet*

(*p_r_*) que contiene a la trama recibida, se crea una nueva trama y se la envía a la capa física (ver el Código Protocolo 4.3).

```
void Protocolo4Agent::recv(Packet* p_r_, Handler*)
{
    // Función que imprimirá en pantalla
    // el nombre del nodo y el tiempo
    registro_evento();

    // Con el Packet p_r_ se accede a frame(hdr_frame)
    r_ = frame::access(p_r_);
    printf(" Recibe la trama (seq,ack):(%d,%d)\n",r_->seq_var,
    r_->ack_var);

    // Se verifica si cumple alguna de las dos condiciones
    if(r_->seq_var==frame_expected ||r_->ack_var==next_frame_to_send)
    {

    /*Verifica si el número de secuencia de la trama
    recibida coincide con el número de la trama que
    esperaba recibir; si ésto se cumple envía el paquete
    a la capa de red e incrementa el número de la
    trama esperada*/

          if(r_->seq_var==frame_expected)
          {
                // Se envía el paquete a la capa de red
                to_network_layer(&r_->info_var);

                // Se cambia el valor de la trama esperada
                inc(frame_expected);
          }

    /*Verifica si el número del ack de la trama recibida
    coincide con el número de secuencia de la última trama
    que fue enviada, si esto se cumple se detiene el
    temporizador para la trama, se obtiene un nuevo
    paquete de la capa de red y se cambia el valor de
    la siguiente trama que se va a enviar
    */

          if(r_->ack_var==next_frame_to_send)
          {     delete p_s_;

                //Se detiene el temporizador
                stop_timer();

                // Se obtiene un nuevo paquete de la capa de red
                from_network_layer(&buffer);

                // Se invierte el número de secuencia del emisor
                inc(next_frame_to_send);
          }
    }
    // Se borra la trama recibida
    delete p_r_;

    p_s_=allocpkt();
```

```
    // Con el Packet p_s_ se accede a frame(hdr_frame)
    s_ = frame::access(p_s_);

    // Se inserta el número de secuencia en la trama
    s_->seq_var=next_frame_to_send;

    // Se inserta el número de confirmación de la trama
    s_->ack_var=1-frame_expected;

    // Se coloca el paquete en el campo info de la trama
    s_->info_var=buffer;

    // Con el Packet p_s_ se accede a hdr_cmn
    hdr_cmn *ch = hdr_cmn::access(p_s_);

    // Se asigna el tamaño de la trama
    ch->size()=frameSize_;

    // Se crea un objeto Packet que será enviado
    Packet *copia;

    // Se crea una copia del p_s_ para enviar
    copia=p_s_->copy();

    // Se envía una copia a la capa física
    to_physical_layer(copia);

    // Se inicia el temporizador
    start_timer();
}
```

Código Protocolo 4.3 Implementación de la función *recv()* de la clase *Protocolo4Agent.*

1.8.1.2. Configuración del escenario de simulación

```
#=========================================================#
# Parámetros que ingresará el usuario                     #
#=========================================================#
set variable(propagacion_) 50       ;# Tiempo de propagación en el
                                    ;# canal en ms
set variable(vtx_)         10       ;# Velocidad de transmisión en Mbps
set variable(frameSize_)   1000     ;# Tamaño de la trama en bytes
set variable(nam)          out4.nam ;# Archivo de trazas .nam
set variable(tr)           out4.tr  ;# Archivo de trazas .tr
set variable(timer_)       0.12     ;# Tiempo para que el temporizador
                                    ;# expire
set variable(loss_)        0.01     ;# Tasa de pérdida de tramas
set variable(iniciar)      0.0      ;# Inicio de la transmisión
set variable(detener)      1.0      ;# Fin de la simulación
#=========================================================#
# Se crea un objeto de la clase Simulator
set ns [new Simulator]

# Diferentes colores para los flujos de datos
$ns color 1 Blue
$ns color 2 Red
```

```
# Creación del archivos de trazas .nam
set nf [open $variable(nam) w]
$ns namtrace-all $nf

# Creación del archivo de trazas .tr
set ntr [open $variable(tr)  w]
$ns trace-all $ntr

# Definición del procedimiento finish que es invocado para
# finalizar la simulación
proc finish {} {
     global ns nf ntr
     $ns flush-trace
     close $nf
     close $ntr
     exit 0
}

# Procedimiento para imprimir el nombre del nodo y el tiempo en el
# que ocurre un evento
Agent/Protocolo4 instproc registro_evento {time} {
     $self instvar node_
     puts "-----------------------------------"
     puts "NODO:[$node_ id], TIME: $time ms"
}
# Creación de los nodos emisor(n0) y receptor(n1)
set n0 [$ns node]
set n1 [$ns node]

# Configuración del color a los nodos
$n0 color blue
$n1 color red

# Creación de los canales que enlazan los nodos
$ns duplex-link $n0 $n1 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail

# Creación del agente y configuración de los parámetros para el
# nodo emisor
set A [new Agent/Protocolo4]
$A set frameSize_ $variable(frameSize_)
$A set timer_ $variable(timer_)

# Creación del agente y configuración de los parámetros para el
# nodo receptor
set B [new Agent/Protocolo4]
$B set frameSize_ $variable(frameSize_)
$B set timer_ $variable(timer_)

# Asignación de los agentes a los nodos
$ns attach-agent $n0 $A
$ns attach-agent $n1 $B

# Conexión entre los agentes
$ns connect $A $B

# Se define el color a los flujos de datos
$A set fid_ 1
$B set fid_ 2
```

```
#-----------------------------------------------#
#                CANAL CON RUIDO                #
#-----------------------------------------------#

# Procedimiento que crea un modelo de pérdida en el canal
proc canal_perdida {tasa_perdida nodo0 nodo1} {
     global ns
     set mod_perdida [new ErrorModel]

     # Taza de pérdida en el canal
     $mod_perdida set rate_ $tasa_perdida ;

     # Se configura la variable aleatoria para
     # la generación de pérdida en el canal
     $mod_perdida ranvar [new RandomVariable/Uniform]

     # El agente Null recibirá los paquetes perdidos
     # en el canal
     $mod_perdida drop-target [new Agent/Null]

     # Enlace al cual se aplicará el modelo de pérdida
     $ns lossmodel $mod_perdida $nodo0 $nodo1
}


# Asignación de un modelo de pérdida para cada conexión
canal_perdida $variable(loss_) $n1 $n0
canal_perdida $variable(loss_) $n0 $n1

# Inicio de la transmisión
$ns at $variable(iniciar) "$A iniciar_transmision "

# Fin de la simulación
$ns at $variable(detener) "finish"

$ns run
```

Script Protocolo4 Configuración del escenario para simular el protocolo de “ventana corrediza de un bit”.

En el Script Protocolo4 se configura el escenario para simular el protocolo de “ventana corrediza de un bit” cuando el agente A empieza con la transmisión. Para que los agentes A y B empiecen con la transmisión simultáneamente, se deberá añadir en el *script* la siguiente línea:

```
$ns at $variable(iniciar) "$B iniciar_transmision "
```

Además, en éste protocolo se ha configurado el mismo tamaño de la trama en los dos agentes que participan en la simulación. En el caso en el que se configura el *script* en el que los dos agentes empiecen con la transmisión simultáneamente se ha configurado una tasa de error igual a 0, ya que si se configura un valor diferente, no se podrá apreciar funcionamiento en el escenario propuesto.

1.8.1.3. Resultados de la simulación

1.8.1.3.1. Impresión en pantalla

Agente A empieza con la transmisión

A continuación en la Figura 3.30, se presentan los resultados que se imprimen en pantalla al realizar la simulación del protocolo “ventana corrediza de un bit” cuando el agente A (asignado al nodo 0) empieza con la transmisión.

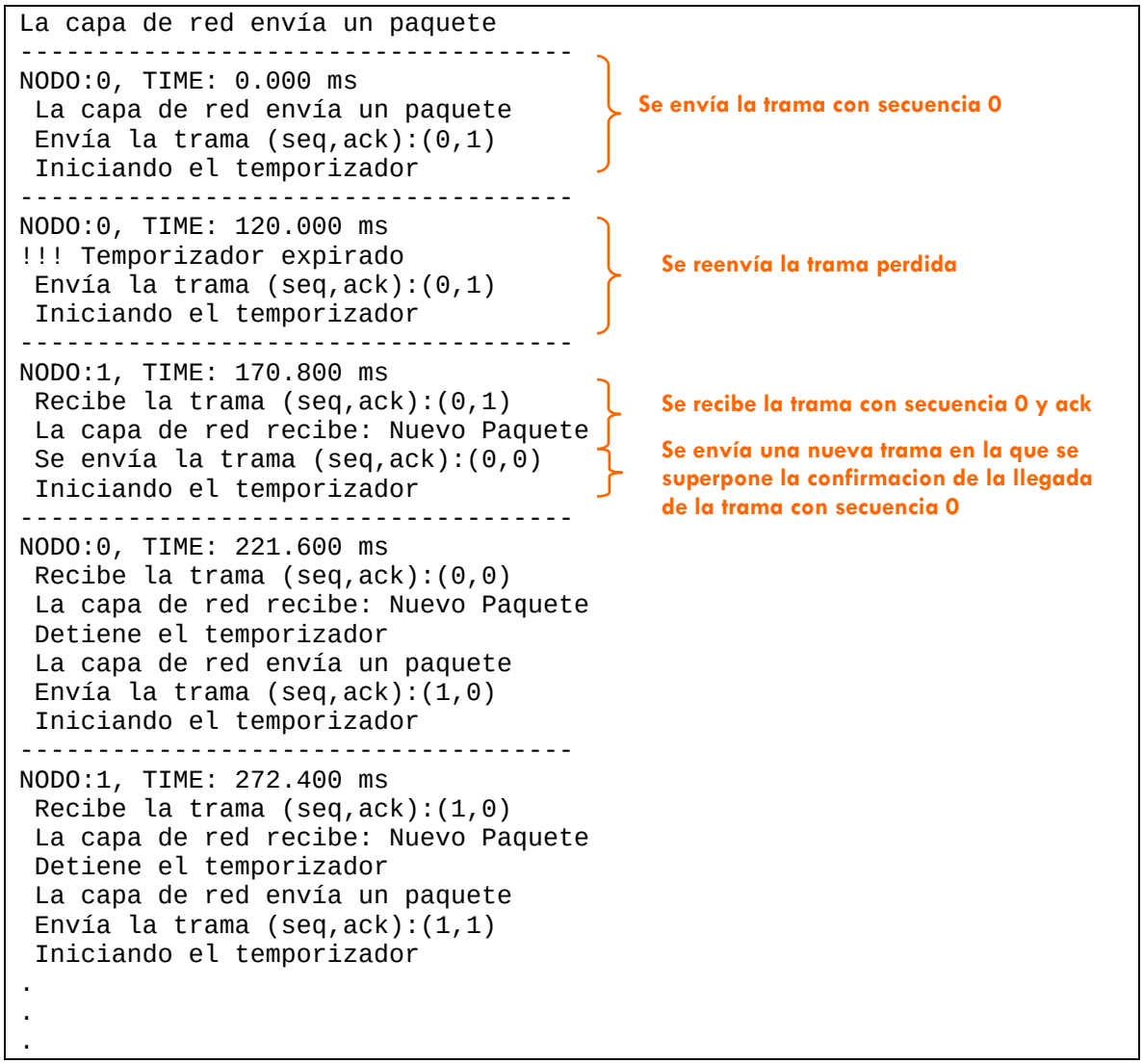


Figura 3.30 Parte de los resultados que se imprimen en pantalla cuando el agente A empieza con la transmisión.

En la Figura 3.30 se puede observar que en una misma trama se transmite datos y control representados en los campos *seq* y *ack* de la trama (por ejemplo al tiempo 0s se transmite la trama con secuencia 0 y acuse de recibo para la trama

1), de igual manera que en el protocolo 3 se inicializa un temporizador por cada trama enviada.

Agentes A y B empiezan simultáneamente con la transmisión

Figura 3. 31 Resultados impresos en pantalla obtenidos de la simulación del protocolo de “ventana corrediza de un bit” para el caso en el que los agentes A y B empieza con la transmisión simultáneamente.

En la Figura 2.31 se puede observar que el agente A (asignado en el nodo 0) y el agente B (asignado en el nodo 1) empiezan con la transmisión al mismo tiempo es decir a los 0ms, lo que provoca que reciba tramas duplicadas las cuales serán

descartadas; por ejemplo al tiempo 50.8ms el agente A recibe una trama con número de secuencia 0 que es la que esperaba, por tanto obtiene el paquete y lo pasa a la capa de red; posteriormente, al tiempo 101.6ms el agente A vuelve a recibir una trama con secuencia 0 por lo que la descarta ya que es un duplicado.

1.8.1.3.2. Archivo out4.tr

Agente A empieza con la transmisión

En la Figura 3.32 se presentan los resultados que se registran en el archivo out4.tr cuando el agente A empieza con la transmisión.

```
d 0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
+ 0.12 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
- 0.12 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
r 0.1708 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
+ 0.1708 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
- 0.1708 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
r 0.2216 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
+ 0.2216 0 1 frame 1000 ------- 1 0.0 1.0 -1 2
- 0.2216 0 1 frame 1000 ------- 1 0.0 1.0 -1 2
r 0.2724 0 1 frame 1000 ------- 1 0.0 1.0 -1 2
+ 0.2724 1 0 frame 1000 ------- 2 1.0 0.0 -1 3
- 0.2724 1 0 frame 1000 ------- 2 1.0 0.0 -1 3
```

El nodo 0 envía una trama al nodo1

El nodo 1 recibe la trama y genera otra trama en la que superpone la confirmación de receptción

Figura 3.32 Parte de los eventos registrados en el archivo out4.tr cuando el agente A empieza con la transmisión.

Agentes A y B empiezan simultáneamente con la transmisión

En la Figura 3.33 se presentan los resultados que se registran en el archivo out4.tr cuando el agente A y B empiezan simultáneamente con la transmisión.

```
+ 0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
- 0 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
+ 0 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
- 0 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
r 0.0508 0 1 frame 1000 ------- 1 0.0 1.0 -1 0
+ 0.0508 1 0 frame 1000 ------- 2 1.0 0.0 -1 2
- 0.0508 1 0 frame 1000 ------- 2 1.0 0.0 -1 2
r 0.0508 1 0 frame 1000 ------- 2 1.0 0.0 -1 1
+ 0.0508 0 1 frame 1000 ------- 1 0.0 1.0 -1 3
- 0.0508 0 1 frame 1000 ------- 1 0.0 1.0 -1 3
r 0.1016 1 0 frame 1000 ------- 2 1.0 0.0 -1 2
```

El nodo 0 envía su primera

El nodo 1 envía su primera

El nodo 1 recibe la trama que envía el nodo 0

El nodo 1 recibe la trama que envía el nodo 0

Figura 3.33 Parte de los eventos registrados en el archivo out4.tr cuando los agentes A y B empieza con la transmisión simultáneamente.

1.8.1.3.3. Archivo out4.nam

Agente A empieza con la transmisión

A continuación en la Figura 3.34, Figura 3.35, Figura 3.36, Figura 3.37 se presentan los resultados que se visualizan al ejecutar el archivo out4.nam obtenido de la simulación del protocolo de “ventana corrediza de un bit” cuando el agente A empieza con la transmisión.

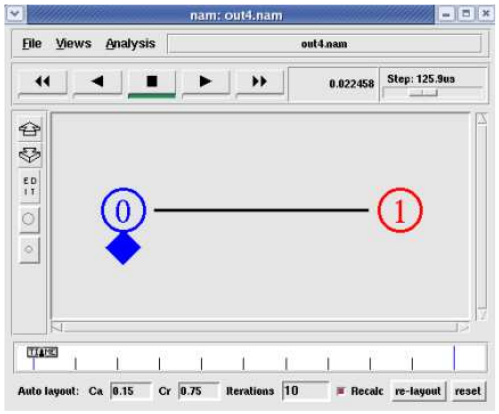


Figura 3.34 Pérdida de la primera trama que intenta transmitir el nodo 0.

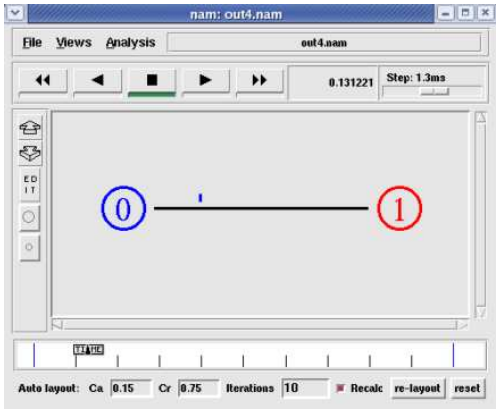


Figura 3.35 El nodo 0 retransmite la trama perdida.

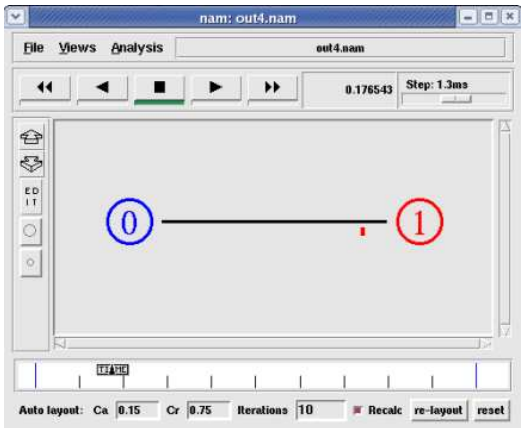


Figura 3.36 El nodo 1 envía la trama con la confirmación superpuesta.

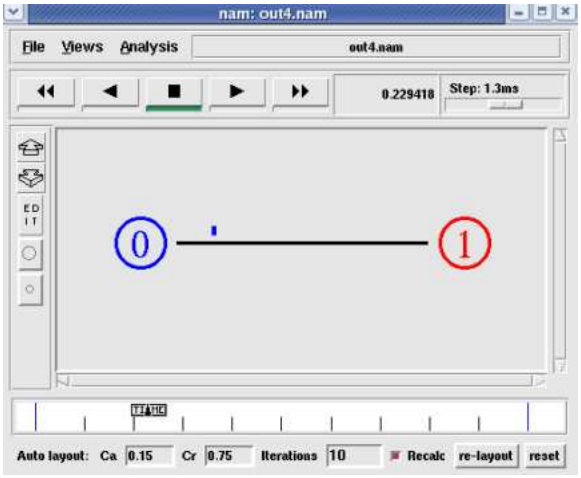


Figura 3.37 El nodo 0 envía una nueva trama.

Agentes A y B empiezan simultáneamente con la transmisión

A continuación en la Figuras 3.38 y la Figura 3.39 se presentan los resultados que se visualizan con la simulación del protocolo “ventana corrediza de un bit” cuando los agentes A y B inician con la transmisión simultáneamente.

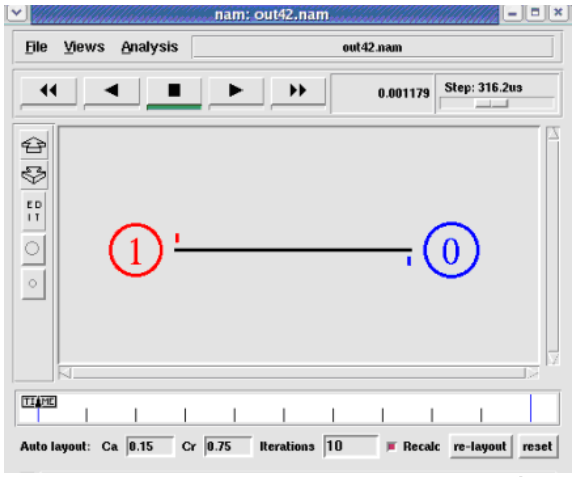


Figura 3.38 Los nodos 0 y 1 empiezan con la transmisión simultáneamente.

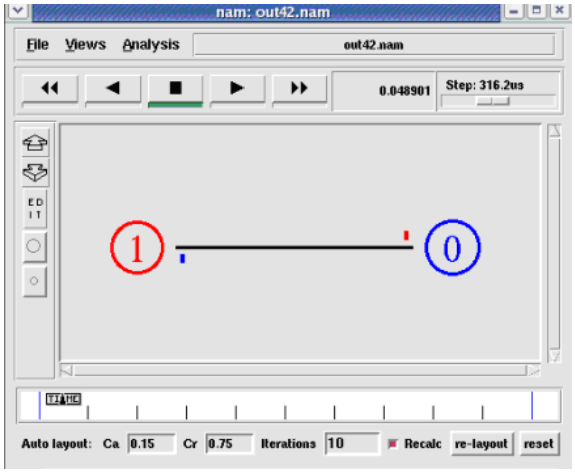


Figura 3.39 Los nodos 0 y 1 reciben las tramas al mismo tiempo.

De los resultados presentados se puede concluir que el funcionamiento del protocolo es el esperado según los escenarios descritos en la Sección 2.2.1.1, así como también se puede observar la técnica de *piggybacking* implementada.

1.8.2. PROTOCOLO DE VENTANA CORREDIZA CON RETROCESO N

1.8.2.1. Diseño e implementación en C++

Una manera para solventar el problema que surge cuando se pierde una trama, consiste en que el receptor simplemente descarte todas las tramas subsecuentes, sin enviar confirmaciones de recepción para las tramas descartadas.

Esta estrategia corresponde a una ventana de recepción de tamaño 1, en otras palabras, la capa de enlace de datos se niega a aceptar cualquier trama, excepto la siguiente que debe entregarse a la capa de red.

El emisor proveerá un búfer en el que almacenará los paquetes contenidos en las tramas transmitidas que aún no se hayan confirmado; si el búfer se llena antes de terminar el temporizador, el canal comenzará a vaciarse. En algún momento, el emisor terminará de esperar y retransmitirá en orden las tramas cuya confirmación no han llegado.

Al igual que en el protocolo de “ventana corrediza de un bit”, que implementa una transmisión bidireccional, se ha considerado implementar un solo agente que cumpla con los roles de emisor y receptor; esto se lo ha realizado mediante la clase *Protocolo5Agent*.

Debido a que este protocolo tiene múltiples tramas pendientes, necesita lógicamente múltiples temporizadores, uno por cada trama pendiente **[7].**

Los temporizadores se han representado con objetos de la clase *Timeout_5.*

Para representar los eventos que envía la capa de red a la capa de enlace de datos en indicación de que tiene nuevos paquete para enviar, también se han utilizado temporizadores representados por objetos de la clase *NetwokLayerReady*.

1.8.2.1.1. Arreglos de temporizadores

Para la implementación de arreglos de temporizadores se ha utilizado el contenedor *vector* de la librería STL[1] **[15, 16].**

En esta implementación se ha considerado necesaria la utilización de un vector que contenga objetos de la clase *Timeout_5* y otro que contenga objetos de la clase *NetwokLayerReady*, como se indica a continuación:

```
typedef vector<Timeout_5 >TimeoutVector;
typedef vector<NetworkLayerReady >NetworkLayerReadyVector;
```

Adicionalmente, para identificar a los vectores se les ha asignado el nombre *TimeoutVector* y *NetworkLayerReadyVector*, mediante el especificador *typedef*.

Para la utilización de los vectores STL es necesario incluir la librería vector *(#include <vector>).*

1.8.2.1.2. Temporizadores

Como se mencionó anteriormente, en este protocolo se implementan dos tipos de temporizadores representados por las siguientes clases:

- ***Timeout_5:*** Cuya definición e implementación es similar a los temporizadores descritos en los protocolos anteriores, por tanto no se profundizará en su explicación. Cada objeto de esta clase representa un temporizador que es asociado a cada trama que se envíe.
- ***NetworkLayerRead:*** Cada objeto de esta clase representa un evento que envía la capa de red a la capa de enlace de datos en comunicación de

[1] *Standard Template Library.*

que tiene nuevos paquetes para enviar. Ante la ocurrencia de este tipo de evento, se invocará automáticamente la función *expire()* cuya implementación se presenta en el Código Protocolo 5.1. Dentro de la función *expire()* se invocará a la función *case_network_layer_ready()* de la clase *Protocolo5Agent* que se explicará más adelante.

```
void NetworkLayerReady::expire(Event*)
{
     networklayerready_->registro_evento();
     printf(" Llega el evento NetworkLayerReady\n");

     // Se invoca a la función case_network_layer_ready()
     // de la clase Protocolo5Agent
     networklayerready_->case_network_layer_ready();
}
```

Código Protocolo 5.1 Implementación de la función *expire()* de la clase *NetworkLayerReady*.

1.8.2.1.3. Agente

Clase Protocolo5Agent

El objetivo de esta clase es proveer funcionalidad para:

- Proveer funciones que indiquen a la capa de red cuando está lista o no para recibir nuevos paquetes.
- Generar y esperar (en orden) la llegada de tramas de datos y de control (objetos de la clase *Packet*).
- Iniciar un temporizador (objetos de la clase *Timeout_5*) por cada una de las tramas enviadas y retransmitir la trama para cual el temporizador expiro y las consecutivas a ésta.

En el Código Protocolo 5.2 se presenta la definición de la clase *Protocolo5Agent.*

```
#define MAX_SEQ 7      // Máximo número que tendrá la secuencia
typedef int seq_nr;    // Número de secuencia o de confirmación

class Protocolo5Agent: public Agent{
private:
// Variables del protocolo
     seq_nr next_frame_to_send;    // Variable que representa
                                   // el número de secuencia de
                                   // la siguiente trama de salida
     seq_nr ack_expected;          // Variable que representa la
                                   // trama más vieja hasta el
                                   // momento no confirmada
```

```
    seq_nr frame_expected;        // Variable que representa
                                  // al número de secuencia de
                                  // la trama que se espera recibir
    frame *s_,*r_;                // Variables que representan a
                                  // las tramas que serán enviadas
                                  // y recibidas respectivamente
    paquete buffer[MAX_SEQ+1];    // Búfer para el flujo de salida
    seq_nr nbuffered;             // Número de búferes de salida
                                  // actualmente en uso
    Packet *p_s_;                 // Variable con la que se podrá
                                  // acceder a frame(hdr_frame)
    int frameSize_;               // Tamaño de la trama
    int i_event;                  // Contador de eventos de tipo
                                  // network_layer_ready
    double vtx_;                  // Velocidad de transmisión del canal

// Variables para los temporizadores
    double timer_;                // Tiempo en el que expirará el
                                  // temporizador
    double TimeEventBefore_;      // Almacena el tiempo programado
                                  // para un evento de tipo
                                  // NetworkLayerReady

// Vector STL de objetos de tipo NetworkLayerReady
    NetworkLayerReadyVector vectorNetworkLayerReady;
    NetworkLayerReadyVector::iterator theIteratorNetworkLayerReady;
    NetworkLayerReady event_network_layer_ready;

// Vector STL de objetos Timeout_5
    TimeoutVector vectorTimer;
    Timeout_5 event_timeout;
    TimeoutVector::iterator theIteratorTimer;

public:
// Constructor
    Protocolo5Agent();
// Destructor
    ~Protocolo5Agent();
// Funciones del protocolo
    void inicialize();            // Empieza con la transmisión
                                  // de datos
    void from_network_layer(paquete * msg)    // Obtiene un paquete
                                              // de la capa de red
    void to_network_layer(paquete *msg);      // Envía un paquete a
                                              // la capa de red
    void to_physical_layer(Packet * psend);   // Envía una trama
                                              // hacia la capa física
    // Función que permite verificar si los valores recibidos
    // caen dentro del rango de valores esperados
    bool between(seq_nr a,seq_nr b,seq_nr c);

    // Elabora y envía una trama de datos
    void send_data(seq_nr frame_nr,seq_nr frame_expected,
    paquete buffer[]);

// Funciones de la clase Agent que se redefinen
    int command (int argc, const char*const* argv);
    void recv(Packet* p_r_, Handler*);
```

```
// Funciones para los temporizadores
    void start_timer(seq_nr k);   // k hace referencia al temporizado
                                  // que corresponde a la trama de
                                  // un número de secuencia dado
    void stop_timer(seq_nr k);    // k hace referencia al ack_expected
    void enable_network_layer();  // Función que permite indicar
                                  // a la capa de red que pueden
                                  // enviar nuevos paquetes
    void disable_network_layer();// Función que indica a la capa
                                  // de red que no puede recibir
                                  // más paquetes
// Funciones auxiliares
    void inc(int& k);             // Incrementa circularmente
                                  // el valor de la trama que está
                                  // esperando enviar o recibir
    void crear_paquete(paquete *msg);   // Genera un paquete en la
                                        // capa de red
    void case_network_layer_ready();    // Función que es invocada
                                        // cuando llega un evento
                                        // network_layer_ready
    void case_timeout();                // Retransmite una trama cuando
                                        // se expira el temporizador
    void registro_evento();             // Función que invoca un
                                        // procedimiento OTcl para
                                        // imprimir  el nombre del nodo
                                        // y el  tiempo en que ocurre
                                        // un evento
    int get_ack_expected();       // Obtiene el número de secuencia
                                  // de la trama de la cual se
                                  // espera la confirmación
};
```

Código Protocolo 5.2 Definición de la clase *Protocolo5Agent*.

Variables

- **seq_nr ack_expected**: Variable que representa a la trama más antigua, hasta el momento no confirmada.
- **nbuffered:** Representa el número de búferes de salida actualmente en uso.
- **i_event:** Representa el identificador del último evento que se ha enviado a la capa de red; su valor se incrementará de manera circular de acuerdo al tamaño de la ventana.
- **buffer[MAX_SEQ + 1]**: Arreglo estático que almacena los paquetes obtenidos de la capa de red cuyo tamaño depende del valor de la ventana, ya que en caso de que se pierdan o dañen, las tramas que se encuentran dentro de la ventana del emisor, se podrán recuperar los paquetes para generar nuevas tramas y retransmitirlas.

- **vectorNetworkLayerReady:** Vector que contiene objetos de la clase *NetworkLayerReady*.
- **theIteratorNetworkLayerReady:** Iterador con el cual se podrá acceder a un objeto de tipo *NetworkLayerReady* que se encuentra en una posición dada del vector *vectorNetworkLayerReady*.
- **event_network_layer_ready:** Objeto de la clase *NetworkLayerReady* del cual se crean copias donde cada una es añadido en cada una de las posiciones del vector *vectorNetworkLayerReady*
- **vectorTimer:** Vector que contiene objetos de la clase *Timeout_5*.
- **event_timeout:** Objeto de la clase *Timeout_5* del cual se crean copias donde cada una es añadido en cada una de las posiciones del vector *vectorTimer*.
- **theIteratorTimer:** Iterador con el cual se podrá acceder a un objeto de tipo *Timeout_5* que se encuentra en una posición dada del vector *vectorTimer.*

Funciones

- **Protocolo5Agent():** Constructor en el que se inicializan los vectores *vectorNetworkLayerReady* y *vectorTimer* con un tamaño igual al de la ventana. En el Código Protocolo 5.3 se presenta su implementación.

```
Protocolo5Agent::Protocolo5Agent() : Agent(PT_FRAME),
event_network_layer_ready(this), event_timeout(this)
{

    // Inicialización de variables
    // Utilizado para el flujo de salida
    next_frame_to_send=0;
    // La trama mas vieja hasta el momento no confirmada
    ack_expected=0;
    // Siguiente trama para el flujo de salida
    frame_expected=0;
    // Número de búfer de salida actualmente en uso
    nbuffered=0;
    // Inicializa en número de eventos network_layer_ready
    i_event=0;
    // Tiempo de llegada del primer evento de
    // tipo NetworkLayerReady
    TimeEventBefore_=0.0;
```

```
// Variables inicializadas desde el archivo .tcl
     bind("timer_", &timer_);
     bind("frameSize_", &frameSize_);
     bind("vtx_",&vtx_);

     // El número del tamaño del vector para los eventos
     // enable_layer_ready será igual al tamaño del
     // número máximo de secuencia

// Inicialización del vector de eventos network_layer_ready
     for(int j=0; j<(MAX_SEQ+1);j++)
     {
           vectorNetworkLayerReady.push_back(event_network_layer_ready);
     }
     theIteratorNetworkLayerReady=vectorNetworkLayerReady.begin();

// Inicialización del vector de temporizadores
     for(int k=0; k<(MAX_SEQ+1);k++)
     {
           vectorTimer.push_back(event_timeout);
     }
     theIteratorTimer=vectorTimer.begin();

}
```

Código Protocolo 5.3 Implementación del constructor de la clase *Protocolo5Agent*.

- **Initialize():** Función que será invocada para empezar con la transmisión de datos. Dentro de ella se invocará a la función *enable_network_layer().*

- **between():** Función que será utilizada para verificar si los valores recibidos caen dentro del rango de valores esperados. En la implementación del protocolo, la función *between()* es utilizada para confirmar si el acuse de recibo coincide con el valor de la siguiente trama que se va a enviar; adicionalmente, se confirma implícitamente a todas las tramas que aún no han sido confirmadas. En el Código Protocolo 5.4 se indica la implementación de esta función.

```
bool Protocolo5Agent::between(seq_nr a,seq_nr b,seq_nr c)
{
     if(((a<=b)&&(b<c)) || ((c<a)&&(a<=b)) || ((b<c)&&(c<a)))
           return (true);
     else
           return (false);
}
```

Código Protocolo 5.4 Implementación de la función *between()* de la clase *Protocolo5Agent*.

- **send_data():** Función en la que se estructura una trama (se asignan los valores a los campos) y se la envía a la capa física. Su implementación se la presenta en el Código Protocolo 5.5.

```
void Protocolo5Agent::send_data(seq_nr frame_nr,seq_nr
frame_expected,paquete buffer[])
{
     // Reserva un espacio de memoria para el puntero
     // p_s_ y configura alguno de los campos para
     // la estructura hdr_cmd
     p_s_=allocpkt();

     // Con el Packet p_s_ se accede a frame(hdr_frame)
     s_ = frame::access(p_s_);

     // Se coloca el paquete en el campo de datos de la trama
     s_->info_var=buffer[frame_nr];
     // Se inserta el número de secuencia en la trama
     s_->seq_var=frame_nr;
     // Se inserta el número de confirmación de la trama
     s_->ack_var=((frame_expected + MAX_SEQ)%(MAX_SEQ+1));
     hdr_cmn *ch = hdr_cmn::access(p_s_);
           // Se asigna el tamaño de la trama
     ch->size()=frameSize_;

     // Se crea un objeto Packet que será enviado
     Packet *copia;
     // Se crea una copia de p_s_ para enviar
     copia=p_s_->copy();
     // Se envía una copia a la capa física
     to_physical_layer(copia);

     // Se inicia el temporizador para la trama
     // con número de secuencia frame_nr
     start_timer(frame_nr);
}
```

Código Protocolo 5.5 Implementación de la función *send_data()* de la clase *Protocolo5Agent*.

- **enable_network_layer():** Función que es utilizada para indicar a la capa de red que recibir un nuevo paquete; esta será invocada cada vez que se haga la verificación del búfer y éste no se encuentre lleno; en esta función se reprogramará un temporizador de tipo *NetworkLayerReady* que expirará dependiendo del valor de la variable *delay_*.

  El valor de la variable *delay_*.es igual a un valor aleatorio más el tiempo de transmisión (*tx_*) para evitar el encolamiento de las tramas; adicionalmente, se hace una comprobación para programar eventos que se consuman progresivamente.

  Para la generación del valor de la variable *delay_*, se utiliza la función *random()* de la clase *Random*. Su implementación se la presenta en el Código Protocolo 5.6.

```
void Protocolo5Agent::enable_network_layer()
{    double tx_;   // Variable que almacena el tiempo de transmisión
     double delay_;// Variable que indica el tiempo en que la capa de
                   // enlace de datos puede recibir un nuevo paquete

do
{    // Se obtiene el tiempo de transmisión de acuerdo al tamaño de la
     // trama y la velocidad de transmisión configurada por el usuario
     tx_=((double)frameSize_*8)/(vtx_*1000000);

     // Se configura el retardo para la llegada de los
     // evetos network_layer_ready
     delay_=((Random::random()%20)*0.0001)+tx_ ;

     // Se hace un comprobación para que los eventos programados se
     // consuman progresivamente, adicionalmente se evita que exista
     // encolamiento
     if((Scheduler::instance().clock()+delay_)>TimeEventBefore_
     && ((Scheduler::instance().clock()+delay_)-TimeEventBefore_)>tx_)
     {
     TimeEventBefore_=Scheduler::instance().clock()+delay_;

     // Se programa al evento que será recibido luego de concluir el
     // tiempo determinado en la variable delay_
           theIteratorNetworkLayerReady->resched(delay_);
     // Dependiendo del valor de la variable i_event se recorrerá el
     // vector vectorNetworkLayerReady

           if(i_event==MAX_SEQ)
     // Si se llega al tamaño máximo del vector,el iterador
     // apuntará a la posición inicial del vector
           theIteratorNetworkLayerReady=vectorNetworkLayerReady.begin();

     // Caso contrario el iterador apuntará a la siguiente posición
           else theIteratorNetworkLayerReady++;
           inc(i_event); // Se cambia el valor de i_event
     }
 }
// En caso de que la programación de eventos no sea progresiva,
// se vuelve a reprogramar el evento
while(Scheduler::instance().clock()+delay_<TimeEventBefore_ &&
((Scheduler::instance().clock()+delay_)-TimeEventBefore_)<tx_);
}
```

Código Protocolo 5.6 Implementación de la función *enable_network_layer*() de la clase *Protocolo5Agent*.

- **disable_network_layer():** Función que es utilizada para indicar a la capa de red que no envíe más paquetes a la capa de enlace de datos; por tanto no programa eventos de tipo *NetworkLayerReady*.
- **recv():** Una vez recibido el objeto de tipo *Packet* se procederá a comprobar si éste está con error, para lo cual se accederá a la cabecera común y se obtendrá el valor de la variable *error_* de dicha cabecera mediante la función *error()*, si el objeto está con error, el valor obtenido

con la función será igual a 1 y se descartará la trama, caso contrario (*ch_r->error()==0*) se aceptará la trama y se la procesará; se verificará si el número de secuencia de la trama recibida es la misma que la esperada, de ser así, se obtendrá el paquete del campo de datos, se enviará a la capa de red, y se avanzará el límite inferior de la ventana del receptor. Además se obtendrá el campo correspondiente al acuse de recibo, con el cual se determinará que tramas llegaron correctamente para detener sus temporizadores y disminuir el búfer de paquetes que se enviaron en las tramas.

Finalmente, se verifica si el búfer está lleno, de ser así se invocará a la función *disable_network_layer(),* caso contrario se invocará a la función *enable_network_layer().*

La implementación de la función *recv()* se la presenta en el Código Protocolo 5.7.

```
void Protocolo5Agent::recv(Packet* p_r_, Handler*)
{
     registro_evento();

     // Con p_r_ se accede a frame
     r_ = frame::access(p_r_);

     // Con p_r_ se accede a hdr_cmn
     hdr_cmn *ch_r = hdr_cmn::access(p_r_);

     // Se verifica si la trama recibida tiene error mediante
     // la variable error_ de la estructura hdr_cmn
     if(ch_r->error()==0)
     {
           printf("Recibe la trama(seq,ack):(%d,%d)\n",
           r_->seq_var,r_->ack_var);

           // Se verifica si el número de secuencia de la trama
           // recibida es la misma que la esperada
           if(r_->seq_var==frame_expected)
           {
                 // Envía el paquete a la capa de red
                 to_network_layer(&r_->info_var);

                 // Se cambia el valor de la trama esperada
                 inc(frame_expected);
           }

     // Se confirma si el acuse de recibo cae dentro del rango de
     // número de secuencia de las tramas de las que se espera la
     // confirmación, además se confirma implícitamente la recepción
     // de las tramas anteriores
```

```
            while (between(ack_expected,r_->ack_var,next_frame_to_send))
            {
                  // Una trama menos en el búfer
                  nbuffered=nbuffered-1;

                  // Se detiene el temporizador
                  stop_timer(ack_expected);

                  // Se cambia el valor del ack esperado
                  inc(ack_expected);
            }
      }

      else
      {
            printf(" Llegó una trama con error \n");
      }

      // Se borra el Packet recibido que contiene a la trama
      delete p_r_;

      printf(" El tamaño del búfer es: %d\n",nbuffered);

      // Si el búfer aun no está lleno, se le comunica a la capa
      // de red para que pueda enviar mas paquetes
      if(nbuffered<MAX_SEQ)

            enable_network_layer();

      // Si el búfer está lleno, se le comunica a la capa
      // de red que no envíe mas paquetes
      else

            disable_network_layer();

}
```

Código Protocolo 5.7 Implementación de la funcion *recv()* de la clase *Protocolo5Agent*.

- **case_network_layer_ready():** Función que será invocada por la función *expire()* de la clase *NetworkLayerReady* cuando un temporizador programado expire; en esta función se verifica si el búfer está lleno, de ser así no se permite programar más temporizadores *network_layer_ready*, caso contrario se obtiene un nuevo paquete de la capa de red, se expande la ventana del emisor, se crea y envía una nueva trama; finalmente, se invoca a la función *enable_network_layer().*

En el Código Protocolo 5.8 se presenta su implementación.

```
void Protocolo5Agent::case_network_layer_ready()
{
     // Si el búfer aun no está lleno, acepta y guarda un nuevo paquete,
     // posteriormente genera y transmite una trama
     if(nbuffered<MAX_SEQ)
     {     // Obtiene un nuevo paquete de la capa de red
           from_network_layer(&buffer[next_frame_to_send]);

           // Expande la ventana del emisor (tamaño de búfer
           // actualmente ocupado)
           nbuffered=nbuffered+1;

           // Trasmite la trama
           send_data(next_frame_to_send,frame_expected,buffer);

           // Avanza el límite superior de la ventana
           inc(next_frame_to_send);

           // Permite eventos network_layer_ready
           enable_network_layer();
     }
     else
           // Si el búfer esta lleno no permite eventos
           // network_layer_ready
           disable_network_layer();
     printf(" El tamaño del búfer es: %d\n",nbuffered);
}
```

Código Protocolo 5.8 Implementación de la función *case_network_layer_ready()* de la clase *Protocolo5Agent.*

- **case_timeout():** Función que será invocada por la función *expire()* de la clase *Timeout_5* cuando un temporizador programado expire; para proceder a reenviar todas las tramas desde aquella a la que se le expiró el temporizador hasta la última que se envió. En el Código Protocolo 5.9 se presenta su implementación.

```
void Protocolo5Agent::case_timeout()
{    // Se asigna el valor de la trama desde la cual
     // se expiró el temporizador
     next_frame_to_send=ack_expected;

     // Lazo que permitira enviar todas las tramas de las que no se ha
     // recibido confiramción
     for(int i=1;i<=nbuffered;i++)
     {     // Detiene el temporizador
           stop_timer(next_frame_to_send);
           // Reenvía la primera trama
           send_data(next_frame_to_send,frame_expected,buffer);
           // Se prepara par enviar la siguente trama
           inc(next_frame_to_send);
     }
}
```

Código Protocolo 5.9 Implementación de la función *case_timeout ()* de la clase *Protocolo5Agent*.

1.8.2.2. Configuración del escenario de simulación

```
# ========================================================#
# Parámetros que ingresará el usuario                     #
# ========================================================#
set variable(propagacion_) 20       ;# Tiempo de propagación en el
                                    ;# canal en ms
set variable(vtx_)         1        ;# Velocidad de transmisión en Mbps
set variable(frameSize_)   1000     ;# Tamaño de la trama en bytes
set variable(nam)          out5.nam ;# Archivo de trazas .nam
set variable(tr)           out5.tr  ;# Archivo de trazas .tr
set variable(timer_)       0.10     ;# Tiempo para que el temporizador
                                    ;# expire
set variable(loss_)        0.01     ;# Tasa de pérdida de tramas
set variable(err_)         0.01     ;# Tasa de tramas con error
set variable(iniciar)      0.0      ;# Inicio de la transmisión
set variable(detener)      0.5      ;# Fin de la simulación
#=========================================================#
# Se crea un objeto de la clase Simulator
set ns [new Simulator]

$ns color 1 Blue
$ns color 2 Red

# Creación del archivo de trazas .nam
set nf [open $variable(nam) w]
$ns namtrace-all $nf

# Creación del archivo de trazas .tr
set ntr [open $variable(tr) w]
$ns trace-all $ntr

# Definición del procedimiento finish que es invocado para
# finalizar la simulación
proc finish {} {
     global ns nf ntr
     $ns flush-trace
     close $nf
     close $ntr
     exit 0
}
# Procedimiento para imprimir el nombre del nodo y el tiempo en el
# que ocurre un evento
Agent/Protocolo5 instproc registro_evento {time} {
     $self instvar node_
     puts "-----------------------------------"
     puts "NODO:[$node_ id], TIME: $time ms"
}
# Creación de los nodos emisor(n0) y receptor(n1)
set n0 [$ns node]
set n1 [$ns node]

# Configuración del color a los nodos
$n0 color blue
$n1 color red

# Creación del canal bidireccional que enlaza a los nodos
$ns duplex-link $n0 $n1 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail
```

```
# Creación del agente y configuración de sus parámetros para el nodo
emisor
set A [new Agent/Protocolo5]
$A set frameSize_ 1000
$A set timer_ $variable(timer_)
$A set vtx_ $variable(vtx_)
# Creación del agente y configuración de sus parámetros para el nodo
# receptor
set B [new Agent/Protocolo5]
$B set frameSize_ 1000
$B set timer_ $variable(timer_)
$B set vtx_ $variable(vtx_)
# Asignación de los agentes a los nodos
$ns attach-agent $n0 $A
$ns attach-agent $n1 $B
# Enlace de los agentes
$ns connect $A $B
# Se define el color a los flujos de datos
$A set fid_ 1
$B set fid_ 2
#-----------------------------------------------------#
#               CANAL CON RUIDO                       #
#-----------------------------------------------------#
# Procedimiento que crea un modelo de pérdida en el canal
proc canal_perdida {tasa_perdida nodo0 nodo1} {
     global ns
     set mod_perdida [new ErrorModel]

     # Taza de pérdida en el canal
     $mod_perdida set rate_ $tasa_perdida;

     # Se configura la variable aleatoria para
     # la generación de pérdida en el canal
     $mod_perdida ranvar [new RandomVariable/Uniform]

     # El agente Null recibirá los paquetes perdidos en el canal
     $mod_perdida drop-target [new Agent/Null]

     # Enlace al cual se aplicará el modelo de pérdida
     $ns lossmodel $mod_perdida $nodo0 $nodo1
}
# Procedimiento que crea un modelo de error en el canal
proc canal_error {tasa_error nodo0 nodo1} {
     global ns
     set mod_error [new ErrorModel]

     # Taza de error en el canal
     $mod_error set rate_ $tasa_error

     # Variable con la cual se pondrá error en el canal
     $mod_error set markecn_ 1

     # Se configura la variable aleatoria para
     # la generación de error en el canal
     $mod_error ranvar [new RandomVariable/Uniform]

     # Enlace al cual se aplicará el modelo de error
     $ns lossmodel $mod_error $nodo0 $nodo1
}
```

```
# Asignación de un modelo de pérdida para cada conexión
canal_perdida $variable(loss_) $n1 $n0
canal_perdida $variable(loss_) $n0 $n1

# Asignación de un modelo de error para cada conexión
canal_error $variable(err_) $n1 $n0
canal_error $variable(err_) $n0 $n1

# Inicio de la transmisión
$ns at $variable(iniciar) "$A iniciar_transmision"

# Fin de la simulación
$ns at $variable(detener) "finish"

$ns run
```

Script Protocolo5 Configuración del escenario para simular el protocolo de "ventana corrediza con retroceso N".

A diferencia de los *scripts* de simulación para los protocolos anteriores, en éste se ha configurado tanto un modelo de error como un modelo de pérdida.

En el modelo de error se debe configurar el valor de la variable *markecn_(1),* además en éste no es necesario la asignación de un agente nulo, ya que los unidades de datos seran recibidas por el agente respectivo.

```
# Procedimiento que crea un modelo de error en el canal
proc canal_error {tasa_error nodo0 nodo1} {
      global ns
      set mod_error [new ErrorModel]

      # Taza de error en el canal
      $mod_error set rate_ $tasa_error

      # Variable con la cual se pondrá error en el canal
      $mod_error set markecn_ 1

      # Se configura la variable aleatoria para
      # la generación de error en el canal
      $mod_error ranvar [new RandomVariable/Uniform]

      # Enlace al cual se aplicará el modelo de error
      $ns lossmodel $mod_error $nodo0 $nodo1
}
```

1.8.2.3. Resultados de la simulación

1.8.2.3.1. Impresión en pantalla

A continuación en la Figura 3.40 se presentan parte de los resultados obtenidos de la simulación del protocolo de "ventana corrediza con retroceso N".

```
-----------------------------------
NODO:0, TIME: 9.600 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (seq,ack) : (0,7)
 Iniciando el temporizador 0
 El tamaño del búfer es: 1
-----------------------------------
NODO:0, TIME: 19.100 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (seq,ack) : (1,7)
 Iniciando el temporizador 1
 El tamaño del búfer es: 2
-----------------------------------
.
.
.
-----------------------------------
NODO:1, TIME: 37.600 ms
 Recibe la trama(seq,ack):(0,7)
 La capa de red recibe:  Nuevo Paquete
 El tamaño del búfer es: 0
-----------------------------------
NODO:0, TIME: 46.200 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (seq,ack) : (4,7)
 Iniciando el temporizador 4
 El tamaño del búfer es: 5
-----------------------------------
NODO:1, TIME: 46.800 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (seq,ack) : (0,0)
 Iniciando el temporizador 0
 El tamaño del búfer es: 1
-----------------------------------
NODO:1, TIME: 47.100 ms
 Recibe la trama(seq,ack):(1,7)
 La capa de red recibe: Nuevo Paquete
 El tamaño del búfer es: 1
-----------------------------------
.
.
.
.
. -----------------------------------
NODO:0, TIME: 137.800 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama con (seq,ack): (5,6)
 Iniciando el temporizador 5
 El tamaño del búfer es: 7
.
.
.
.
```

El nodo emisor envía tramas

El nodo receptor recibe la trama

Se envía una trama superponiendo la confirmación

Se tranmite una trama con secuencia 5

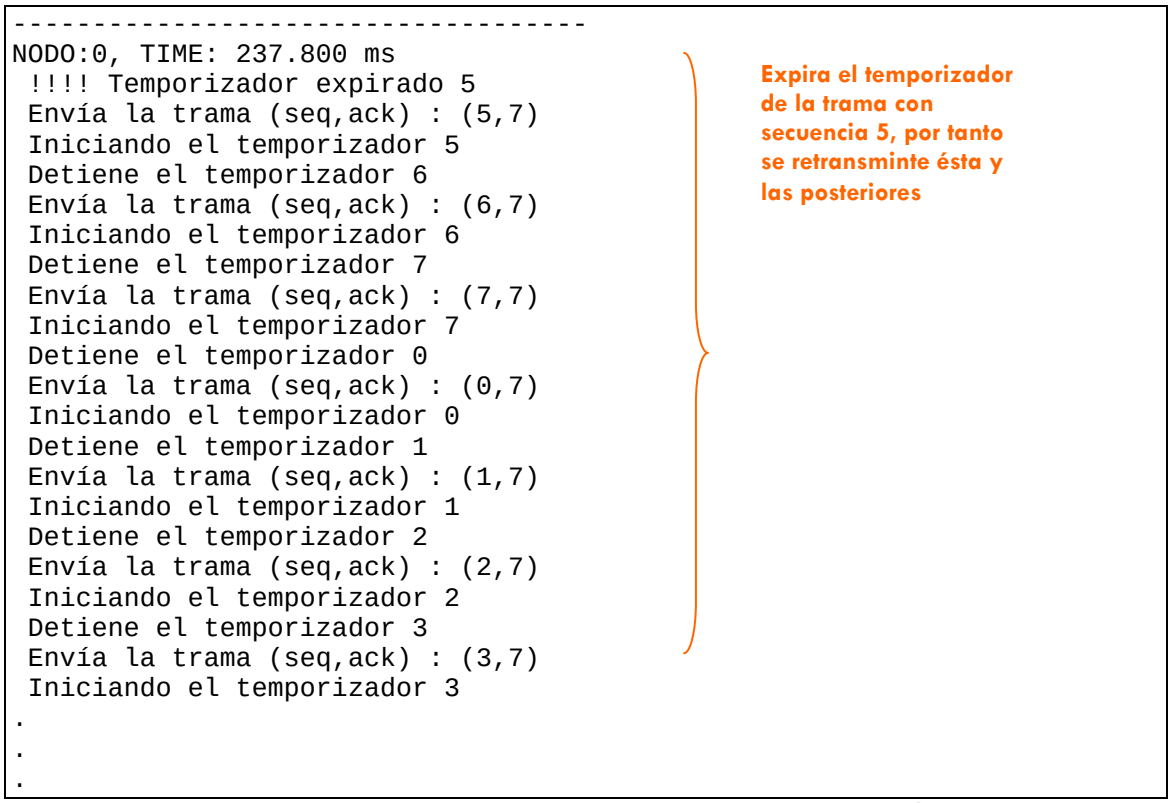


Figura 3.40 Resultados impresos en pantalla obtenidos de la simulación del protocolo de "ventana corrediza con retroceso N".

En los resultados que se presentan en la Figura 3.40 se puede apreciar:

- Cuando un módulo envía correctamente una trama y ésta a su vez es recibida (por ejemplo al tiempo 9.6ms el nodo 0 envía la trama (0,7) y ésta es recibida por el nodo 1 al tiempo 37.6ms).
- Cuando una trama se pierde, por tanto no se envía la confirmación para ella, por lo que expira el temporizador correspondiente (por ejemplo al tiempo 137.8ms la trama que es enviada por el nodo 0 se pierde y al tiempo 237.8ms expira el temporizador correspondiente) y se reenvían las tramas para las cuales no ha llegado la confirmación.

1.8.2.3.2. Archivo out5.tr

En la Figura 3.41 se presentan los resultados que se registran en el archivo out5.tr.

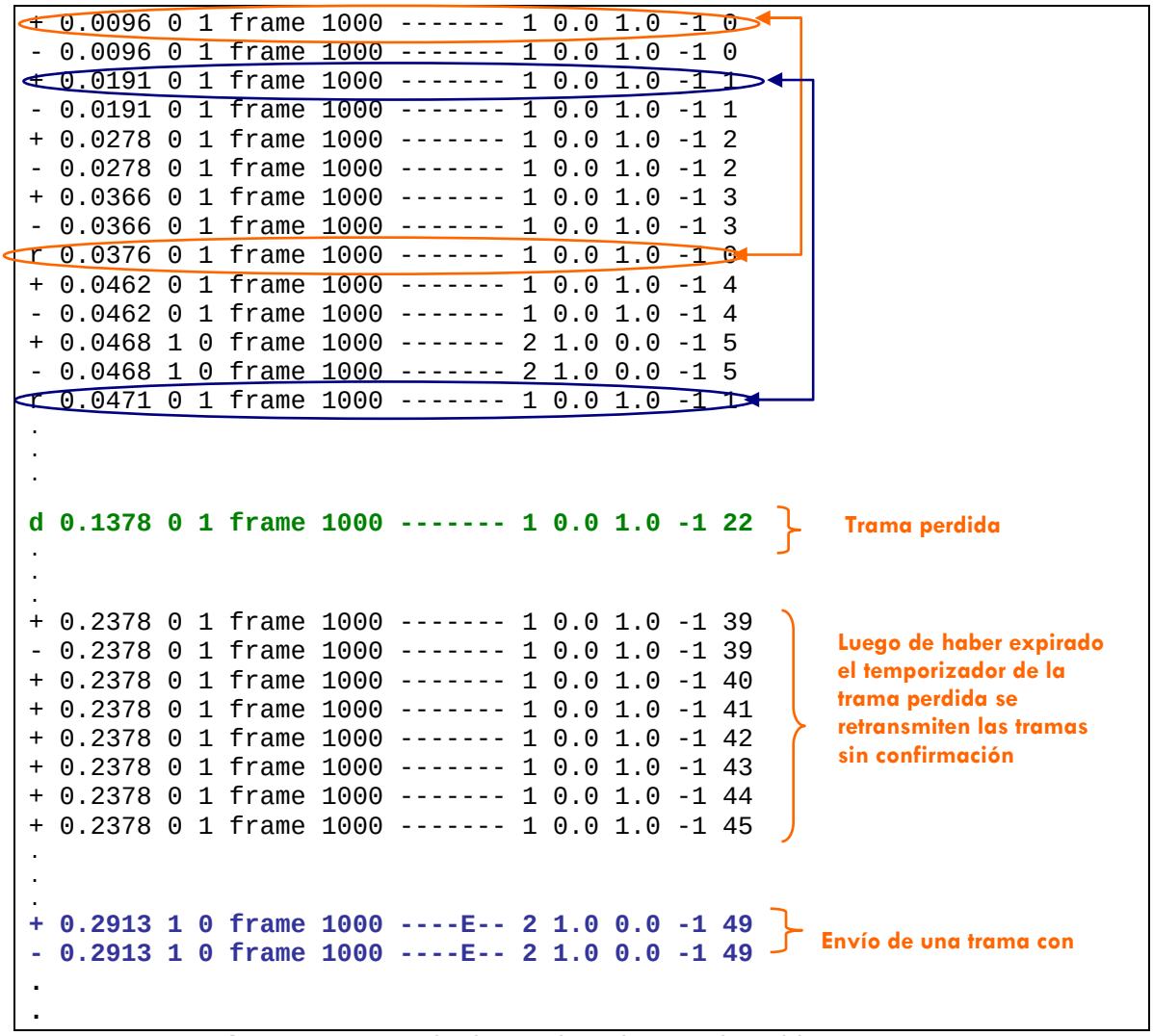


Figura 3.41 Resultados registrados en el archivo out5.tr.

Las tramas con error se transmitirán de forma similar que las tramas sin error, en el archivo .tr se las puede distinguir ya que presentan la bandera (*E*) como se puede observar al tiempo 0.2913 s.

1.8.2.3.3. Archivo out5.nam

A continuación en la Figura 3.42, Figura 3.43, Figura 3.44, Figura 3.47 se presentan los resultados que se visualizan al ejecutar el archivo ou5.nam.

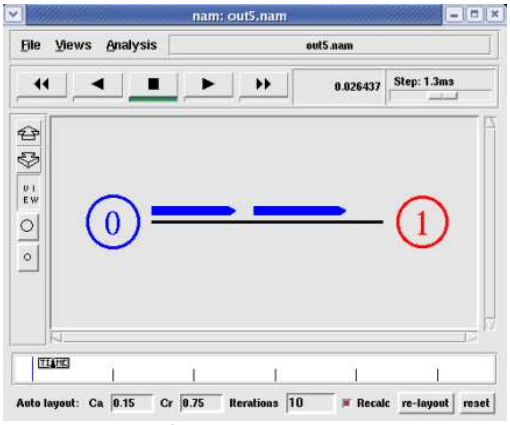


Figura 3.42 Inicio de la transmisión, en donde el nodo 0 envía tramas al nodo 1.

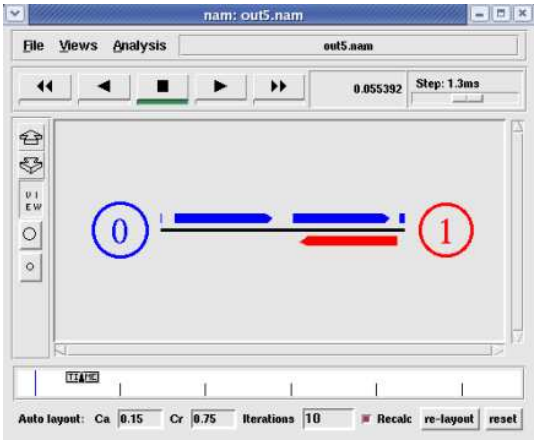


Figura 3.43 El nodo 1 recibe las tramas y envía las confirmaciones superpuestas en nuevas tramas.

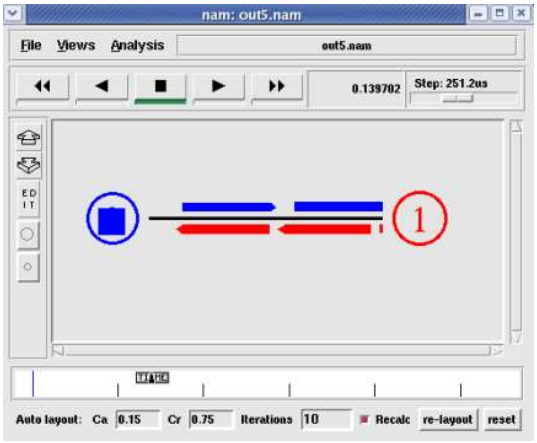


Figura 3.44 El nodo 0 pierde una trama.

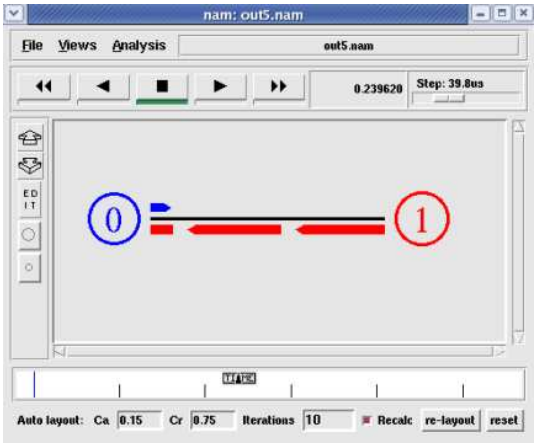


Figura 3.45 Expira un temporizador en el nodo 0 y se retransmiten las tramas pendientes.

Con la ejecución del protocolo de “de ventana corrediza con retroceso N” se ha obtenido resultados con los que se verifica que están acorde a las consideraciones descritas para el mismo, es así que se puede observar que en comparación con los protocolos anteriores, en éste se aprovecha la utilización del canal de comunicación ya que no permanece desocupado mientras espera por la confirmación de las tramas enviadas, sin embargo se puede apreciar también la desventaja que presenta el protocolo al tener que reenviar todas las tramas posteriores a una trama cuyo temporizador ha expirado.

1.8.3. PROTOCOLO DE VENTANA CORREDIZA DE REPETICIÓN SELECTIVA

1.8.3.1. Diseño e implementación en C++

Para dar solución al problema descrito en el protocolo de “ventana corrediza de repetición selectiva” y evitar que haya traslape, el tamaño de la ventana deberá ser de al menos la mitad del intervalo de los números de secuencia; por tanto, para números de secuencia de 3 bits, el tamaño de ventana será 4.

De igual manera, el número de búferes necesarios será igual al tamaño de la ventana, no al intervalo de números de secuencia, ya que en ninguna circunstancia puede aceptar tramas cuyos números de secuencia estén por debajo del extremo inferior o por encima del extremo superior de la ventana. Por la misma razón, el número de temporizadores requeridos es igual al número de búferes, no al tamaño del espacio de secuencia; por tanto, hay un temporizador

(representado por la clase *Timeout_6)* asociado a cada búfer. Cuando el temporizador expire, el contenido del búfer se retransmitirá.

Para la retransmisión del paquete contenido en el búfer, al cual está asociado el temporizador, se decidió proveer funcionalidad que permita obtener el número de secuencia de dicha trama y asignarla en una variable (*frame_sec_)* propia del temporizador.

En este protocolo, en caso de que no haya tráfico al cual superponer la confirmación de recepción, se inicializa un temporizador auxiliar (representado por la clase *Ack_Timeout*) tras la llegada de una trama de datos en secuencia, si éste expira antes de que se presente tráfico en la dirección contraria, se envía una trama de confirmación de recepción independiente. Esto evitará una retransmisión por expiración del temporizador asociado a la trama.

Al igual que en los dos protocolos anteriores se ha implementado un solo agente que cumpla con los roles de emisor y receptor, esto se lo ha realizado mediante la clase *Protocolo6Agent*.

La representación de los eventos que sirven para indicar a la capa de enlace de datos que la capa de red tiene nuevos paquetes se ha basado en temporizadores, para lo cual se ha implementado la clase *NetwokLayerReady_6.*

Debido al requerimiento de que haya un temporizador asociado a cada uno de los búferes, se hace uso de arreglos de temporizadores de manera similar a lo que se realizó en el protocolo anterior.

El temporizador auxiliar será representado mediante un objeto de la clase *AckTimeoutTimer*.

1.8.3.1.1. Agente

Clase Protocolo6Agent

El objetivo de esta clase es proveer funcionalidad para:

- Proveer funciones que indiquen a la capa de red cuando está lista o no para recibir nuevos paquetes.
- Generar tramas de datos y de control.

- Esperar la llegada de tramas de datos, obtener los paquetes de ellas y almacenarlos en el búfer aún cuando no estén en orden. Una vez que se obtengan los paquetes en orden enviarlos a la capa de red.
- Iniciar un temporizador (objetos de la clase *Timeout_6*) por cada una de las tramas enviadas y retransmitir sólo la trama de la que expire el temporizador.

En el Código Protocolo 6.1 se presenta la definición de la clase *Protocolo6Agent.*

```
#define MAX_SEQ 7                   // Debe ser 2^n -1 donde n es el
                                    // número de bits
#define NR_BUFS ((MAX_SEQ+1)/2)     // Número de bufers
typedef int seq_nr;
class Protocolo6Agent: public Agent{
private:
// Variables del protocolo
    seq_nr next_frame_to_send;      // Variable que representa
                                    // al número de secuencia de
                                    // la siguiente trama de salida
    seq_nr ack_expected;            // Variable que representa al
                                    // número de confirmación que se
                                    // espera recibir
    seq_nr frame_expected;          // Variable que representa
                                    // al número de secuencia de
                                    // la trama que espera recibir
    frame *r_, *s_;                 // Variables que representan a las
                                    // tramas que serán recibidas y
                                    // enviadas respectivamente
    paquete out_buf[NR_BUFS];       // Arreglo que almacenará los paque-
                                    // tes que van a ser enviados
    paquete in_buf[NR_BUFS];        // Arreglo que almacenará los paque-
                                    // tes que van a ser recibidos
    seq_nr nbuffered;               // Número de búferes de salida que
                                    // se utilicen actualmente
    Packet *p_s_;                   // Variable con la que se podrá
                                    // acceder a frame(hdr_frame)
    int frameSize_;                 // Tamaño de la trama
    bool no_nak;                    // Variable que permitirá indicar si
                                    // se ha enviado una trama de tipo
                                    // nak
    seq_nr oldest_frame;            // Variable que representa el número
                                    // de secuencia de la trama cuyo
                                    // temporizador ha expirado
    seq_nr too_far;                 // Límite superior de la ventana
                                    // del receptor
    bool arrived [NR_BUFS];         // Permite indicar si una posición
                                    // del búfer está llena o vacía
    int i_event;                    // Variable que permitirá verificar
                                    // si el búfer está en posibilidad
                                    // de aceptar nuevos paquetes
    int vtx_;                       // Velocidad de transmisión del canal

// Variables para los temporizadores
```

```
    double timer_;                  // Variable que representa el
                                    // intervalo de tiempo que espera
                                    // antes que el temporizador expire
    double TimeEventBefore_;        // Almacena el tiempo programado
                                    // para un evento de tipo
                                    // NetworkLayerReady

    double ack_timer_out_;         // Variable que representa el inter-
                                   // valo de tiempo que se espera antes
                                   // de enviar una trama de confirmación

// Vector STL de objetos de tipo NetworkLayerReady
    NetworkLayerReady_6 event_network_layer_ready;
    NetworkLayerReadyVector vectorNetworkLayerReady ;
    NetworkLayerReadyVector::iterator theIteratorNetworkLayerReady;

// Vector STL de objetos de tipo Timeout_6
    Timeout_6 event_timeout;
    TimeoutVector vectorTimer;
    TimeoutVector::iterator theIteratorTimer;

// Ack_Timeout
    AckTimeoutTimer event_acktimeout;

public:
// Constructor
    Protocolo6Agent();
// Destructor
    ~Protocolo6Agent();
// Funciones del protocolo
    void initialize();                  // Empieza con la transmisión
                                        // de datos
    void from_network_layer(paquete * msg);   // Obtiene un paquete de
                                              // la capa de red
    void to_network_layer(paquete* msg);      // Envía el paquete a
                                              // la capa de red
    void to_physical_layer(Packet * psend);   // Envía una trama
                                              // hacia la capa física
    // Función que permite verificar si los valores recibidos
    // caen dentro del rango de valores esperados
    bool between(seq_nr a, seq_nr b, seq_nr c);
    // Función que realiza el entramado y envía el trama a la
    // capa física
    void send_frame(frame_kind kind,seq_nr frame_nr,
    seq_nr frame_expected,paquete buffer[]);

// Funciones de la clase Agent que se redefinen
    int command (int argc, const char*const* argv);
    void recv(Packet* pkt, Handler*);

// Funciones para los temporizadores
    void start_timer(int aux_frame_nr); // Inicia el temporizador
                                        // asociado a un búfer
    void stop_timer(int ack_expected);  // Detiene el temporizador
                                        // asociado a un búfer
     void start_ack_timer();       // Inicia un temporizador auxiliar
                                   // dentro de cuyo intervalo se puede
                                   // enviar una trama de confirmación
                                   // independiente
    void stop_ack_timer();         // Detiene el temporizador auxiliar
```

```
    void enable_network_layer( ); // Función que permite indicar
                                  // a la capa de red que puede
                                  // enviar nuevos paquetes
    void disable_network_layer(); // Función que indica a la
                                  // capa de red  que no puede
                                  // recibir más paquetes
// Funciones auxiliares
    void inc(int& k);       // Incrementa circularmente
                            // el tamaño de la ventana
    void inc_aux(int& k);   // Incrementa circularmente el valor búfer
    void case_ack_timeout();// Transmite una trama de confirmación
                            // cuando expira un temporizador de un
                            // ack_timeout
    void case_network_layer_ready(void);     // Función que es invocada
                                             // cuando llega un evento
                                             // network_layer_ready
    void case_timeout(int x);     // Retransmite una trama cuando se
                                  // expira el temporizador
    void crear_paquete(paquete *msg);   // Genera un paquete en la
                                        // capa de red
    void registro_evento();       // Función que invoca un
                                  // procedimiento OTcl para
                                  // imprimir en pantalla el nombre
                                  // del nodo y el tiempo en que
                                  // ocurre un evento en éste
};
```

Código Protocolo 6.1 Definición de la clase *Protocolo6Agent*.

Variables

- **out_buf[NR_BUFS]:** Arreglo estático de tamaño *NR_BUFS* en el que se almacenarán los paquetes obtenidos de la capa de red que han sido enviados en las tramas correspondientes y que aún no han sido confirmadas.
- **in_buf[NR_BUFS]:** Arreglo estático de tamaño *NR_BUFS* en el que se almacenarán los paquetes que han sido obtenidos de los campos de datos de las tramas recibidas.
- **no_nak:** Variable que representa una bandera que es utilizada para indicar si antes se ha enviado una trama de confirmación negativa.
- **oldest_frame:** Variable que representa el número de secuencia de la trama cuyo temporizador ha expirado.
- **too_far:** Variable que indica el límite superior de la ventana del receptor.
- **arrived[NR_BUFS]:** Arreglo estático con el que se indica si una posición del búfer está llena o vacía.

- **ack_timer_out_:** Variable que representa el intervalo de tiempo que se espera para enviar una trama de confirmación independiente.

Funciones

- **send_frame():** Función en la que se realiza el entramado, se verifica si la trama a ser enviada es de datos o de confirmación negativa mediante el campo *kind_var*. Si la trama no es de datos no se envía información útil, caso contrario se almacena en el campo *info_var* el paquete obtenido de la capa de red, se programa un temporizador y se detiene el temporizador auxiliar. Cabe recalcar que el primer argumento es utilizado para diferenciar el tipo de trama, el valor asignado depende del enumerado *frame_kind* definido en la estructura *hdr_frame.* Su implementación se presenta en el Código Protocolo 6.2.

```
void Protocolo6Agent::send_frame(frame_kind fk,seq_nr frame_nr,seq_nr
frame_expected,paquete buffer[])
{    p_s_ =allocpkt();
     s_= frame::access(p_s_);
     // Se asigna el tipo de trama
     s_->kind_var=fk;
     // Si la trama es de datos, se coloca un paquete en
     // el campo de datos
     if(fk==data)
           s_->info_var = buffer[frame_nr%NR_BUFS];
     s_->seq_var=frame_nr;
     s_->ack_var= (frame_expected + MAX_SEQ)%(MAX_SEQ+1);
     // Si la trama es de confirmación negativa se configura
     // la bandera no_nak para indicar que ya se envió
     // una trama de este tipo
     if(fk==nak)
           no_nak=false;
     hdr_cmn *ch = hdr_cmn::access(p_s_);
     // Se asigna el tamaño de la trama
     ch->size()=frameSize_;
     // Se crea un objeto Packet que será enviado
     Packet *copia;
     // Se crea una copia del p_s_ para enviar
     copia=p_s_->copy();
     to_physical_layer(copia);
     if(fk==data)
     {     //Si la trama es de datos se inicia el temporizador
           start_timer(frame_nr);
     }
// Debido a que se envia una trama que contiene la confirmación
// de recepción respectiva, se detiene el temporizador
// de confirmación auxiliar
     stop_ack_timer();
}
```

Código Protocolo 6.2 Implementación de la función *send_frame()* de la clase *Protocolo6Agent*.

- **case_timeout():** Función que será invocada en la función *expire()* de la clase *Timeout_6*. Una vez que un temporizador expira, ésta sabe cual trama retransmitir debido al valor obtenido de su argumento, el cual es otorgado por la función *get_secuencia()* de la clase *Timeout_6*. Su implementación se presenta en el Código Protocolo 6.3.

```
void Protocolo6Agent::case_timeout(int x)
{
     oldest_frame=x;
     send_frame(data,oldest_frame,frame_expected,out_buf);
}
```

Código Protocolo 6.3 Implementación de la función *case_timeout()* de la clase *Protocolo6Agent*.

- **case_ack_timeout():** Función que será invocada en la función *expire()* de la clase *AckTimeoutTimer;* permitirá enviar una trama de confirmación independiente en el caso que no haya una trama de datos en la cual superponer la confirmación. Su implementación se presenta en el Código Protocolo 6.4.

```
void Protocolo6Agent::case_ack_timeout()
{
     printf(" Llega el evento AckTimeout\n");

// Debido a que no es una trama de datos no es necesario asignar
// un número de secuencia, por esta razón se envia el valor 0
     send_frame(ack,0,frame_expected,out_buf);
}
```

Código Protocolo 6.4 Implementación de la función *case_ack_timeout()*de la clase *Protocolo6Agent.*

- **recv():** Función en la que se verifica si el objeto de tipo *Packet* (*p_r_*) llega con error o no. Si llega con error se envía una trama de confirmación negativa, caso contrario se obtiene la trama y se verifica si es de datos o de confirmación negativa.

  Si la trama es de datos se verifica sus campos de acuerdo a los valores esperados. Si el número de secuencia de la trama es diferente a la esperada *(r_−>seq_var!=frame_expecte*d) y no se ha enviado una trama de confirmación negativa para ésta, entonces se procede a enviarla. Si el número de secuencia de la trama recibida está en el rango de valores de las tramas esperadas, se obtendrá el paquete y se almacenará en el

búfer de entrada (*in_buf*) para posteriormente ordenarlos y pasarlos a la capa de red.

Si la trama es de confirmación negativa se procede a enviar lo solicitado en ella, si esta trama se perdiera el funcionamiento del protocolo no se altera.

Posteriormente, se verifica si el valor contenido en el campo de confirmación de la trama está en el rango de valores de confirmación esperados.

Su implementación se presenta en el Código Protocolo 6.5.

```
void Protocolo6Agent::recv(Packet* p_r_, Handler*)
{
     registro_evento();
     // Con p_r_ se accede a frame
     r_ = frame::access(p_r_);

     // Imprimirá en pantalla los siguientes datos:
     // (tipo de trama, número de secuencia, acuse de recibo)
     printf(" Se recibe la trama (kind,seq,ack):(%d,%d,%d)\n",
     r_->kind_var,r_->seq_var,r_->ack_var);

     // Con p_r_ se accede a hdr_cmn
     hdr_cmn *ch_r = hdr_cmn::access(p_r_);

// Se verifica si la trama que llega tiene error
if(ch_r->error_==0)
{
     // Se verifica si es una trama de datos
     if (r_->kind_var==data)
     {
// Se hace una comprobación del número de secuencia de la trama
// recibida
// Pueden presentarse dos casos:
// 1.- La trama no es la esperada; si esto ocurre se envía un nak
// (confirmación de recepción negativa) que indica cual es la trama
// que se está esperando, pero para esto se verifica si antes ya se
// ha enviado una nak para dicha trama, ya que no es la intención
// enviar muchas tramas nak
// 2.- La trama es la esperada

     // Primer caso (La trama no es la esperada)
                 if((r_->seq_var!=frame_expected) && no_nak)
                 {
// Se procede a enviar una trama nak, que indica la trama que no ha
// llegado(frame_expected); al no ser una trama de datos no es
// importante el número de secuencia

                   send_frame(nak,0,frame_expected,out_buf);
                 }
     // Segundo caso (La trama es la esperada)
```

```
                else
                {
// Se inicia un temporizador auxiliar. Si no se ha presentado tráfico
// de regreso antes de que termine este temporizador, se envía una
// trama de confirmación de recepción independiente
                      start_ack_timer();
                }

// Sin importar si el número de secuencia de la trama que llega
// coincide o no con la trama esperada, se realiza la verificación
// para ver si la trama recibida cáe en el rango de los números de
// secuencia de las tramas que se están esperando
// Si lo anterior ocurre se verifica si la posición en el búfer en
// el que se va almacenar la trama no esté ocupado

          if(between(frame_expected,r_->seq_var,too_far)&&
          (arrived[r_->seq_var%NR_BUFS]==false))
          {
// Se indica que la posición en el búfer en donde se almacena la trama
// que ha llegado está ocupada
                arrived[r_->seq_var%NR_BUFS]=true;

// Se obtiene el paquete y se lo almacena en el búfer de entrada
                in_buf[r_->seq_var%NR_BUFS]=r_->info_var;

// Se pasa a la capa de red los paquetes, sólo si están en orden, caso
// contrario permanecerán almacenadas en el búfer hasta que estén en el
// orden esperado
                while(arrived[frame_expected%NR_BUFS])
                {
                     to_network_layer(&in_buf[frame_expected%NR_BUFS]);
                      no_nak=true;

// Una vez que se ha pasado los paquetes a la capa de red se indica
// que el búfer esta en condición de almacenar nuevas tramas entrantes
                      arrived[frame_expected%NR_BUFS]=false;
                      inc(frame_expected);
                      inc(too_far);

// Se inicia un temporizador para enviar una trama de confirmación
// independiente en el caso de que este expire y aún no se haya
// enviado la confirmación
                      start_ack_timer();
                }
          }
    }
// En el caso de que la trama recibida sea una nak, se verifica si el
// valor de confirmación contenida en la trama nak cae
// dentro del rango de valores de las tramas por las que se espera
// la confirmación
    if((r_->kind_var==nak)&& between(ack_expected,
    (r_->ack_var+1)%(MAX_SEQ+1),next_frame_to_send))
    {
// Se envía la trama que está esperando el receptor
                send_frame(data,(r_->ack_var+1)%(MAX_SEQ+1),
                frame_expected,out_buf);
    }
// Se verifica si el valor de confirmación contenida en la trama
// recibida cae dentro del rango de valores de las tramas por las
// que espera confirmación
```

```
          while(between(ack_expected,r_->ack_var,next_frame_to_send))
          {
              nbuffered=nbuffered-1;
// Se detiene el temporizador para la trama cuya confirmación
// se estaba esperando
              stop_timer(ack_expected%NR_BUFS);
// Se cambia el valor de la trama de quien se espera la confirmación
              inc(ack_expected);
              printf(" El valor de ack_expected es:%d\n",ack_expected);
          }
     }
else
{
     // Si llega un trama con error
     printf(" Llego una trama con error\n");
     if(no_nak==true)
     {
          send_frame(nak,0,frame_expected,out_buf);
     }
}
     delete p_r_;
     printf(" El tamaño del búfer es: %d\n",nbuffered);
     // Se verifica si el búfer está lleno
     if(nbuffered<NR_BUFS)
     {
          // Se habilita para recibir un nuevo paquete de la
          // capa de red
          enable_network_layer();
     }
     else
     {    // Se niega a recibir nuevos paquetes de la
          // capa de red
           disable_network_layer();
     }
}
```

Código Protocolo 6.5 Implementación de la función *recv()* de la clase *Protocolo6Agent*.

1.8.3.2. Configuración del escenario de simulación

```
#=========================================================#
# Parámetros que ingresará el usuario                     #
#=========================================================#
set variable(propagacion_) 20       ;# Tiempo de propagación en el
                                    ;# canal en ms
set variable(vtx_)         1        ;# Velocidad de transmisión en Mbps
set variable(frameSize_)   1000     ;# Tamaño de la trama en bytes
set variable(nam)          out6.nam ;# Archivo de trazas .nam
set variable(tr)           out6.tr  ;# Archivo de trazas .tr
set variable(timer_)       0.10     ;# Tiempo para que el temporizador
                                    ;# expire
set variable(ack_timer_out_) 0.010  ;# Tiempo de espera del
                                    ;# temporizador ack
set variable(loss_)        0.01     ;# Tasa de pérdida de tramas
set variable(err_)         0.02     ;# Tasa de tramas con error
set variable(iniciar)      0.0      ;# Inicio de la transmisión
set variable(detener)      0.5      ;# Fin de la simulación
#=========================================================#
```

```
# Se crea un objeto de la clase Simulator
set ns [new Simulator]

$ns color 1 Blue
$ns color 2 Red

# Creación del archivo de trazas .nam
set nf [open $variable(nam) w]
$ns namtrace-all $nf
# Creación del archivo de trazas .tr
set ntr [open $variable(tr) w]
$ns trace-all $ntr

# Definición del procedimiento finish que es invocado para
# finalizar la simulación
proc finish {} {
     global ns nf ntr
     $ns flush-trace
     close $nf
     close $ntr
     exit 0
}
# Procedimiento para imprimir el nombre del nodo y el tiempo en el
# que ocurre un evento
Agent/Protocolo6 instproc registro_evento {time} {
     $self instvar node_
     puts "---------------------------------------"
     puts "NODO:[$node_ id], TIME: $time ms"
}

# Creación de los nodos
set n0 [$ns node]
set n1 [$ns node]
# Configuración del color a los nodos
$n0 color blue
$n1 color red

# Creación del canal bidireccional que enlaza a los nodos
$ns duplex-link $n0 $n1 $variable(vtx_)Mb $variable(propagacion_)ms
DropTail

# Creación del agente y configuración de sus parámetros
# para el nodo emisor
set A [new Agent/Protocolo6]
$A set timer_ $variable(timer_)
$A set ack_timer_out_ $variable(ack_timer_out_)
$A set frameSize_ $variable(frameSize_)
$A set vtx_ $variable(vtx_)

# Creación del agente y configuración de sus parámetros
# para el nodo receptor
set B [new Agent/Protocolo6]
$B set timer_ $variable(timer_)
$B set ack_timer_out_ $variable(ack_timer_out_)
$B set frameSize_ $variable(frameSize_)
$B set vtx_ $variable(vtx_)

# Asignación de los agentes a los nodos
$ns attach-agent $n0 $A
$ns attach-agent $n1 $B
```

```
# Enlace de los agentes
$ns connect $A $B

# Se define el color a los flujos de datos
$A set fid_ 1
$B set fid_ 2
#-------------------------------------------------#
#                CANAL CON RUIDO                  #
#-------------------------------------------------#
# Procedimiento que crea un modelo de pérdida en el canal
proc canal_perdida {tasa_perdida nodo0 nodo1} {
     global ns
     set mod_perdida [new ErrorModel]

     # Taza de pérdida en el canal
     $mod_perdida set rate_ $tasa_perdida ;

     # Se configura la variable aleatoria para
     # la generación de pérdida en el canal
     $mod_perdida ranvar [new RandomVariable/Uniform]

     # El agente Null recibirá los paquetes perdidos en el canal
     $mod_perdida drop-target [new Agent/Null]

     # Enlace al cual se aplicará el modelo de pérdida
     $ns lossmodel $mod_perdida $nodo0 $nodo1
}
# Procedimiento que crea un modelo de error en el canal
proc canal_error {tasa_error nodo0 nodo1} {
     global ns
     set mod_error [new ErrorModel]

     # Taza de error en el canal
     $mod_error set rate_ $tasa_error

     # Variable con la cual se pondrá error en el canal
     $mod_error set markecn_ 1

     # Se configura la variable aleatoria para
     # la generación de error en el canal
     $mod_error ranvar [new RandomVariable/Uniform]

     # Enlace al cual se aplicará el modelo de error
     $ns lossmodel $mod_error $nodo0 $nodo1
}
# Asignación de un modelo de pérdida para cada conexión
canal_perdida $variable(loss_) $n1 $n0
canal_perdida $variable(loss_) $n0 $n1
# Asignación de un modelo de error para cada conexión
canal_error $variable(err_) $n1 $n0
canal_error $variable(err_) $n0 $n1
# Inicio de la transmisión
$ns at $variable(iniciar) "$A iniciar_transmision "
# Fin de la simulación
$ns at $variable(detener) "finish"
$ns run
```

Script Protocolo6 Configuración del escenario para simular el protocolo de “ventana corrediza de repetición selectiva”.

1.8.3.3. Resultados de la simulación

1.8.3.3.1. Impresión en pantalla

A continuación, en la Figura 3.46 se indican los resultados que se imprimen en pantalla al realizar la simulación del protocolo de "ventana corrediza con repetición selectiva".

```
-----------------------------------
NODO:0, TIME: 9.600 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (kind,seq,ack): (0,0,7)
 Iniciando el temporizador de la trama 0
 El tamaño del búfer es: 1
-----------------------------------
NODO:0, TIME: 19.100 ms
 Llega el evento NetworkLayerReady
 La capa de red envía un paquete
 Envía la trama (kind,seq,ack): (0,1,7)
 Iniciando el temporizador de la trama 1
 El tamaño del búfer es: 2
-----------------------------------
.
.
-----------------------------------
NODO:1, TIME: 47.100 ms
 Se recibe la trama (kind,seq,ack):(0,1,7)
 Envía la trama (kind,seq,ack): (2,0,7)
 El tamaño del búfer es: 0
-----------------------------------
.
.
-----------------------------------
NODO:0, TIME: 75.100 ms
 Recibe la trama (kind,seq,ack):(2,0,7)
 Envía la trama (kind,seq,ack): (0,0,7)
 Iniciando el temporizador de la trama 0
 El tamaño del búfer es: 4
 Búfer lleno, no se generan eventos
..
-----------------------------------
NODO:1, TIME: 103.100 ms
 Recibe la trama (kind,seq,ack):(0,0,7)
 Activa el temporizador ack_timeout
 La capa de red recibe:  Nuevo Paquete
 Activa el temporizador ack_timeout
 La capa de red recibe:  Nuevo Paquete
 Activa el temporizador ack_timeout
 La capa de red recibe:  Nuevo Paquete
 Activa el temporizador ack_timeout
 La capa de red recibe:  Nuevo Paquete
 Activa el temporizador ack_timeout
 El tamaño del búfer es: 4
 Búfer lleno, no se generan eventos
```

Llega la trama incorrecta,por tanto se envía al nodo 0 una nak

El nodo 0 recibe nak y reenvia la trama solicitada

El nodo 1 recibe la trama (0,0,7), entonces puede envía los paquetes ordenados de las tramas recibida anteriormente a la capa de red

Figura 3.46 Resultados impresos en pantalla obtenidos de la simulación del protocolo de "ventana corrediza de repetición selectiva".

En los resultados que se presentan en la Figura 3.46, a diferencia de los resultados del protocolo anterior se puede apreciar:

- Cuando se recibe una trama que no es la esperada (o llega una trama con error) se envía una trama de confirmación negativa en solicitud de retransmisión de la trama (por ejemplo al tiempo 47.100ms llega la trama (0,1,7) cuando espera recibir la trama (0,0,7), por tanto se envía la trama (2,0,7); el valor del campo *kind* igual a 2 indica que es una trama de tipo *nak*, el valor de *seq* igual a 0 para este caso indica que no tiene datos y el valor de *ack* igual a 7 indica que la última confirmación recibida es de la trama con *seq* igual a 7).
- Cuando se recibe una *nak* se procede a reenviar la trama solicitada (por ejemplo al tiempo 75.100ms se reenvía la trama (0,0,7); el valor de *seq* es igual a 0, ya que la *nak* indica que la última confirmación recibida es de la trama con *seq* igual a 7),
- Una vez recibida la trama con secuencia correcta, se ordena los paquetes de las trama que se recibió anteriormete y se los pasa a la capa de red (por ejemplo al tiempo 103.100s llega la trama (0,0,7)).

Los resultados de la simulación se los obtiene en los archivos out6.tr y out6.nam.

1.8.3.3.2. Archivo out6.tr

En la Figura 3.47 se presentan los resultados que se registran en el archivo out6.tr.

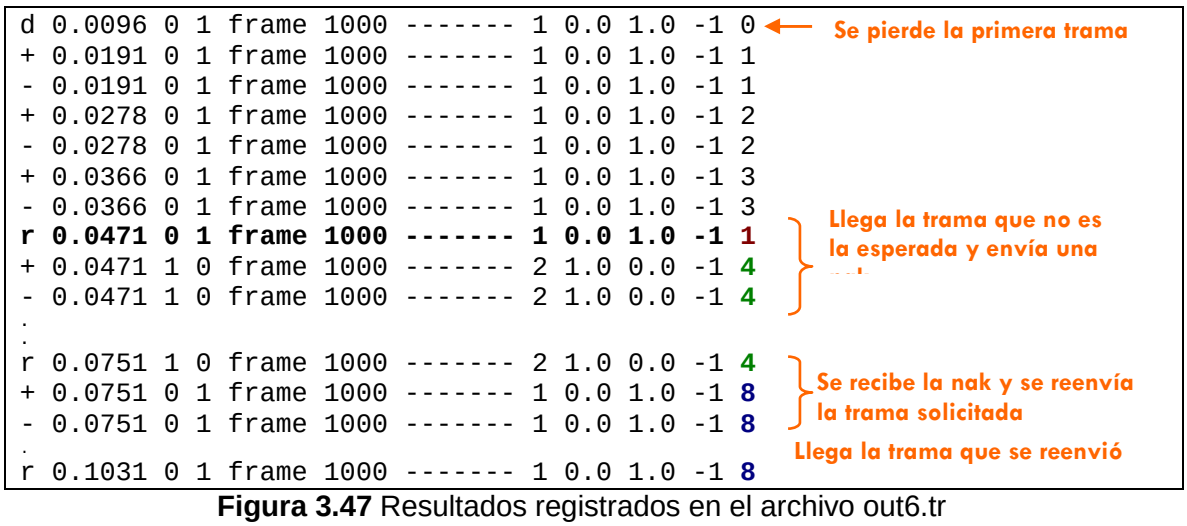


Figura 3.47 Resultados registrados en el archivo out6.tr

1.8.3.3.3. Archivo out6.nam

A continuación se presentan los resultados que se visualizan al ejecutar el archivo out6.nam obtenido de la simulación del protocolo “ventana corrediza de repetición selectiva”.

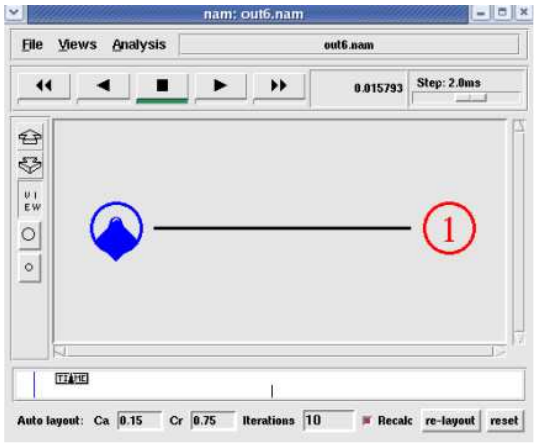


Figura 3.48 Se pierde la primera trama que se intenta transmitir.

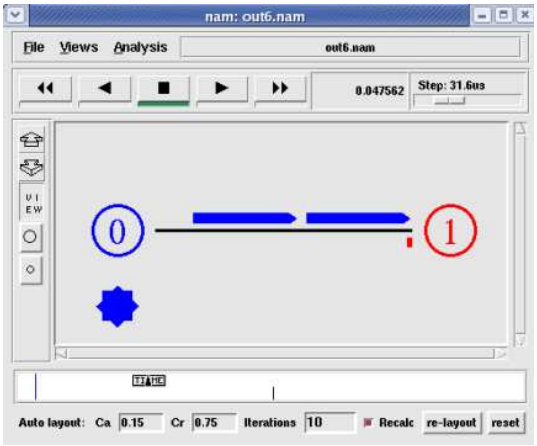


Figura 3.49 La segunda trama enviada llega al destino, pero ya que no es la esperada, se envia una *nak*.

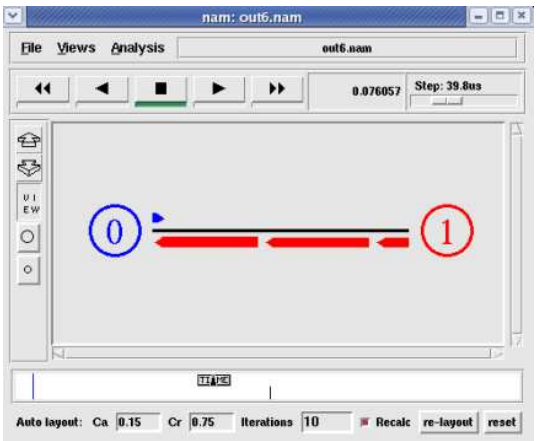


Figura 3.50 Se recibe la *nak* y se reenvía la trama perdida.

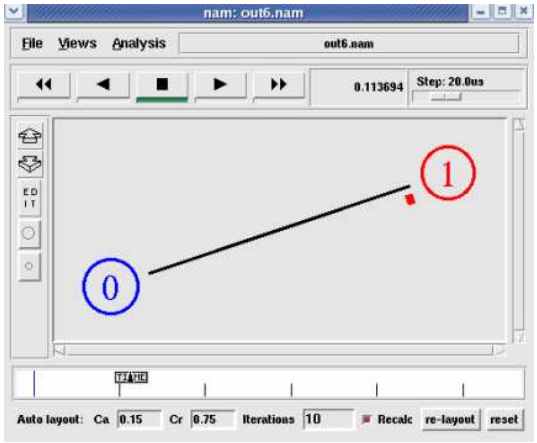


Figura 3.51 Se pasan los paquetes en orden a la capa de red.

De los resultados obtenidos se puede concluir que el protocolo cumple con las consideraciones planteadas para su implementación, es así que si una trama que llega es detectada con error esta es descartada e inmediatamente se informa enviando una trama de control; si la trama se pierde en el canal, el temporizador expira y se procederá a la retransmisión; ya no hay la dependencia de que haya datos en la que se pueda superponer el acuse de recibo, por tanto esta confirmación será independiente y, finalmente cumple con la naturaleza propia del protocolo que es la de retransmisión selectiva.

Con el cumplimiento de esas condiciones se puede garantizar que es el protocolo que mejor ocupa el canal de comunicación.

1.9. INTERACCIÓN DEL SIMULADOR CON NUEVOS PROTOCOLOS IMPLEMENTADOS

Como se explicó en la Sección 1.3, NS-2 posee un intérprete OTcl como interfaz hacia el usuario, en el cual se instanciarán y utilizarán objetos de red, cuya funcionalidad se encuentra implementada en el espacio C++ (por razones de eficiencia). Por tanto, el simulador provee mecanismos para establecer la correspondencia entre estos dos espacios (es decir desde el espacio OTcl hacer uso de la funcionalidad implementada en C++ y viceversa).

El procedimiento que sigue el simulador, para la vinculación entre los objetos de las jerarquías compilada e interpretada, es transparente para el usuario.

1.9.1. PROCESO DE VINCULACIÓN C++/OTCL DE OBJETOS DE RED

Para explicar el proceso de vinculación entre los espacios C++ y OTcl, se ha tomado como ejemplo un objeto de red que provee la funcionalidad de un protocolo, cuya clase dentro del espacio C++ es *ProtocoloAgent*, y su correspondiente clase dentro del espacio OTcl es *Agent/Protocolo*[1]. Cabe indicar que esta sección no tiene como objetivo explicar el funcionamiento característico del objeto de red sino el proceso de vinculación.

Para la utlización de los objetos de red implementados en el espacio C++, desde el espacio OTcl es necesario:

- Definir e implementar la clase que representa el objeto de red (*ProtocoloAgent*).
- Definir e implementar la clase que realiza la vinculación (*ProtocoloClass*).

El la Figura 3.52 presenta la implementación de un objeto de red en el espacio C++.

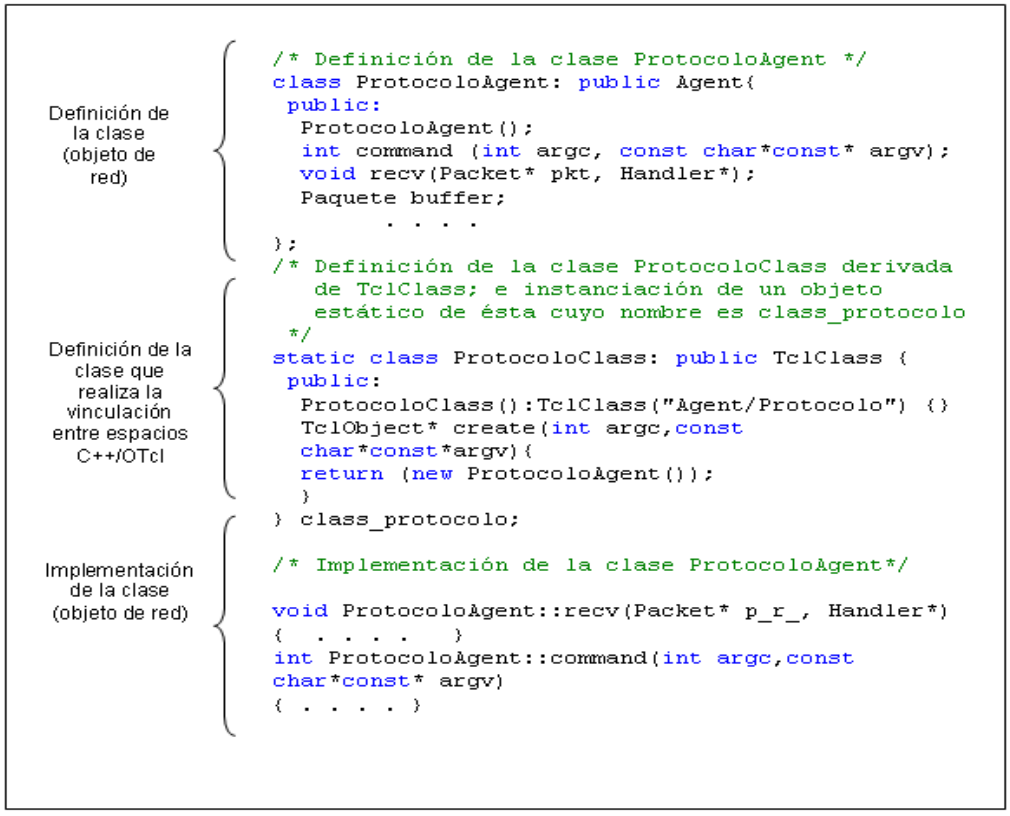


Figura 3.52 Implementación de un objeto de red en el espacio C++.

[1] Clase que será creada por el simulador en el espacio OTcl.

Un objeto de red debe ser ubicado dentro de la jerarquía compilada de acuerdo a su funcionalidad. Para este caso, se lo ha ubicado como se indica en la Figura 3.53.

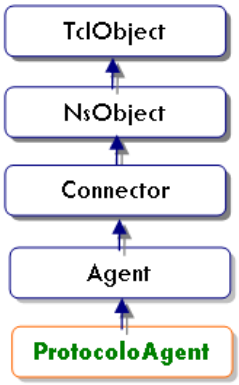


Figura 3.53 Ubicación de la clase *ProtocoloAgent* dentro de la jerarquía compilada.

Una vez definido el objeto de red e implementada su funcionalidad en C++, es necesario que éste se vea reflejado dentro de la jerarquía interpretada, como se ilustra en la Figura 3.54. Este mecanismo lo provee la clase *ProtocoloClass*, la misma que es derivada de la clase *TclClass*.

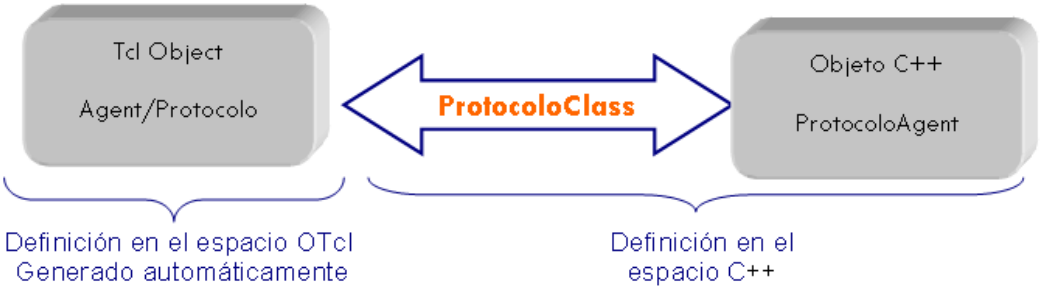


Figura 3.54 Reflejo de un objeto en C++ con su correspondiente en OTcl [**17]**

El proceso de vinculación empieza cuando se ejecuta el simulador NS-2; donde, lo primero que ocurre es la creación de la lista enlazada estática de objetos *TclClass* y de objetos estáticos los cuales son insertados en la lista. Por tanto, haciendo referencia a la Figura 3.52, se inicializa el objeto estático *class_protocolo*, lo cual implica que se llamará al constructor de la clase *ProtocoloClass* y a continuación al constructor de la clase *TclClass*.

El constructor de la clase *TclClass* tomará como argumento el nombre de la clase que será creada en la jerarquía interpretada; es decir, en el ejemplo, tomará como argumento *Agent/Protocolo*[1].

Continuando con el proceso de inicialización del simulador, desde la función *main()* de NS-2 se invoca a la función global *Tcl_AppInit()*[2]*;Tcl_AppInit()* a su vez invoca a la función *bind()* de la clase *TclClass,* para cada objeto estático de la lista enlazada. Esta secuencia de eventos se las puede revisar en la Figura 3.63

En la Figura 3.55 se presenta el código de la función *bind()* de la clase *TclClass*.

```
void TclClass::bind()
{

/* Se obtiene la referencia a la instancia de Tcl */
    Tcl& tcl = Tcl::instance();

/* Mediante la función evalf() se invoca al procedimiento register{} de
   la clase SplitObject para cada objeto, enviándole como argumento
   el nombre de la clase (contenido en la variable classname_) a ser
   registrada en el espacio OTcl.
 */
    tcl.evalf("SplitObject register %s", classname_);

/* Se obtiene un puntero de tipo OTclClass mediante la función
   OTclGetClass() y se lo asigna a class_, a través de éste se crearán
   los procedimientos necesarios para la vinculación como son:
   create_shadow{} y delete_shadow{} para la clase a la que se refiere
   la variable classname_
 */
    class_ = OTclGetClass(tcl.interp(), (char*)classname_);

/* Correspondencia entre los procedimientos y las funciones
 * en los dos espacios
 */
// class_ hace referencia a la jerarquía de clases interpretada

    OTclAddIMethod(class_, "create-shadow",
    (Tcl_CmdProc *) create_shadow, (ClientData)this, 0);

    OTclAddIMethod(class_, "delete-shadow",
    (Tcl_CmdProc *) delete_shadow, (ClientData)this, 0);
}
```

Figura 3.55 Función *bind()* de la clase *TclClass.*

La función *bind()* permite la interacción entre el espacio C++ y OTcl, a través de una referencia al intérprete (*Tcl& tcl = Tcl::instance()*).

---

[1] En NS-2 se utiliza el carácter "/" como un separador. Para cualquier clase A/B/C la clase A/B/C es derivada de la clase A/B y este a su vez es derivada de la clase A.

[2] Función invocada por la función *main()* del simulador.

Con la referencia al intérprete, se invoca a la función *evalf()*, con la cual permite invocar al procedimiento *register{}*[1], implementado en el espacio OTcl, enviando como argumento el nombre de la clase interpretada.

El procedimiento *register{},* de acuerdo al argumento recibido (*Agent/Protocolo*), generará dentro del espacio OTcl clases derivadas de la clase *SplitObject*. En la Figura 3.56 se presenta el código del procedimiento *register{}.*

```
# El procedimiento register{} recibe como argumento el nombre
# de la clase que se creará dentro del espacio OTcl, es decir
# para el ejemplo citado, el argumento será Agent/Protocolo
SplitObject proc register className {
# De acuerdo al argumento recibido (className), se creará la lista
# classes, que contendrá los nombres que se encuentren separados por el
# caracter "/", en este caso, los elementos de la lista serán: Agent y
# Protocolo
set classes [split $className /]
set parent SplitObject
set path "" ;# Representa a la última clase creada con el procedimiento
set sep ""  :# Separador

# Lazo que con el que se recorre los elementos de la lista classes
foreach cl $classes {

    # En la variable cl se almacenarán los valores de los elementos
    # de la lista classes, es decir la primera vez el valor de cl
    # será Agent, la segunda vez será Protocolo
    set path $path$sep$cl
          if ![$self is-class $path] {
          # En caso de no existir la clase con nombre igual al
          # contenido en la variable $path, se creará la clase,
          # la misma que será derivada de la clase $parent
          Class $path -superclass $parent
          }
    # Se configuran nuevos valores para las variables
    set sep /
    set parent $path
}
}
# Deacuerdo al ejemplo en mención se tendrá el siguiente procedimiento:
# Primero:- El valor de la variable path será: Agent
#         - Si existe una clase de nombre Agent no se crea ninguna
#           clase con ese nombre
#         - El nuevo valor de la varible sep es "/",y el nuevo valor
#           de la variable parent es Agent.
# Segundo:- El valor de la variable path será: Agent/Protocolo
#         - Si no exite una clase de nombre Agent/Protocolo, se la crea
#         - El nuevo valor de la varible sep es "/", y el nuevo
#           valor de la variable parent es Agent/Protocolo
# Tercero:- Si no existen más elementos en la lista classes,
#           se termina la ejecución del procedimiento
```

Figura 3.56 Procedimiento *register{}* de la clase *SplitObject* en el espacio OTcl.

[1] Procedimiento de la clase *SplitObject* en el espacio OTcl.

A continuación, la Figura 3.57 presenta el diagrama de flujo para el procedimiento *register{}.*

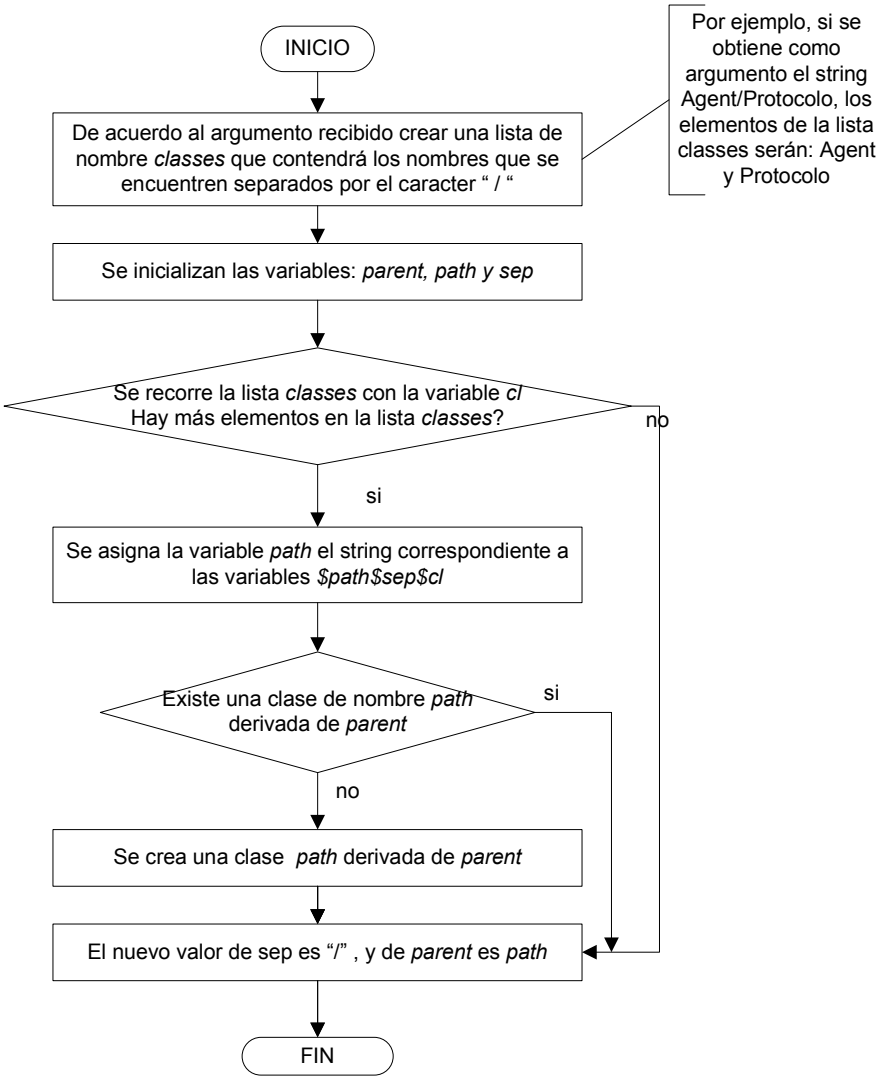


Figura 3.57 Diagrama de flujo de la función *register{}.*

Retomando a la Figura 3.52, se enviará *Agent/Protocolo* como argumento al constructor de la clase *TclClass*, lo que implica que se creará la jerarquía interpretada como se ilustra en la Figura 3.58.

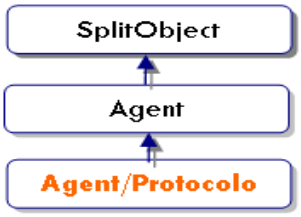


Figura 3.58 Jerarquía interpretada para la clase *Agent/Protocolo*.

A continuación, en la función *bind()* indicada en la Figura 3.55 se invoca la función *OTclGetClass(),* con la cual se obtiene un puntero de tipo *OTclClass* que es asignado a la variable *class_,* a través de ésta se añadirán los procedimientos *create_shadow{}*[1] y *delete_shadow{}* para la clase *Agent/Protocolo,* y los relacionará con las funciones *create-shadow()*[2] *y delete-shadow()* respectivamente.

Una vez creada la clase *Agent/Protocolo,* es posible instanciar objetos de ésta, dentro del *script* de simulación.

En el *script* OTcl, se podrá instanciar objetos de la clase *Agent/Protocolo* (*set protocolo [new Agent/Protocolo]),* lo que implica invocar a su constructor y a los de sus clases base, hasta invocar al constructor de la clase *SplitObject* con la finalidad de inicializar las variables de esta clase, como se presenta en la Figura 3.59.

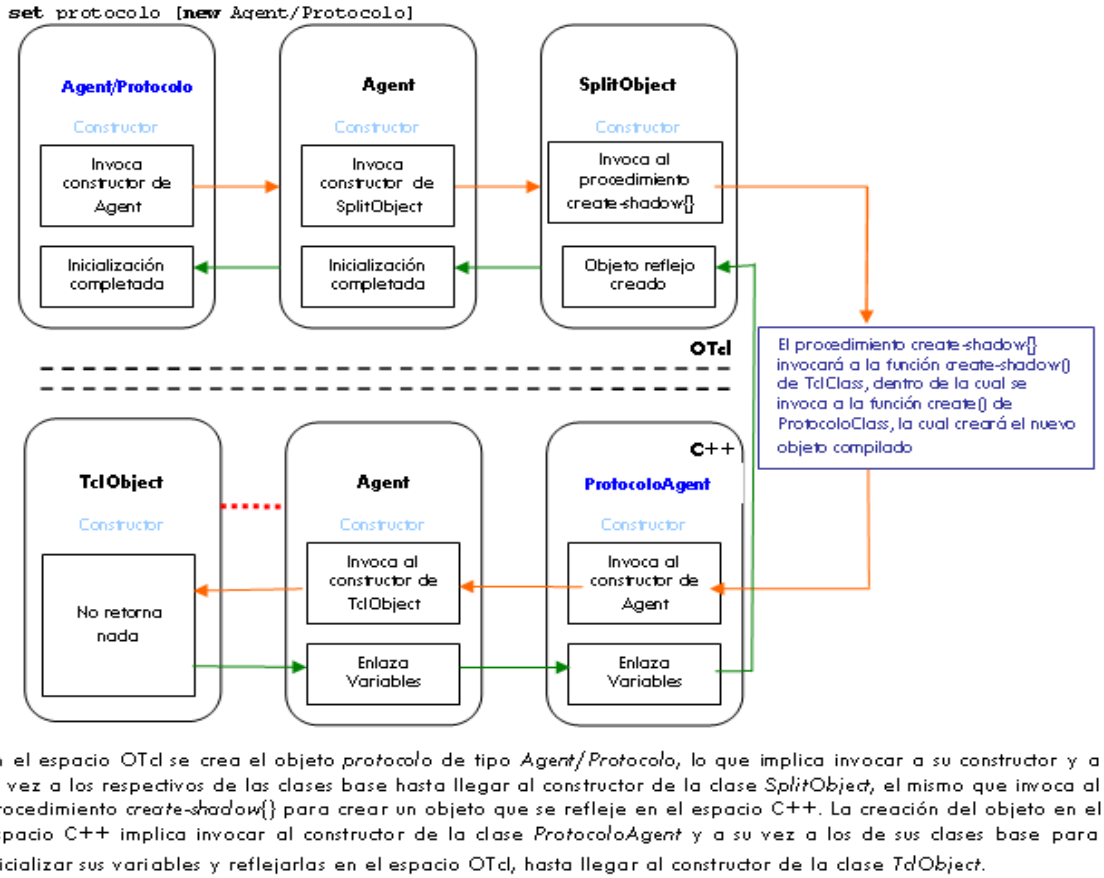


Figura 3.59 Correspondencia de objetos C++ y OTcl **[18]**.

[1] El procedimiento *create_shadow{}* será invocado por el constructor de *SplitObject* al inicializar variables para un objeto.
[2] create_shadow() y *delete_shadow():* funciones de la clase *TclClass.*

El constructor *init{}* de la clase *SplitObject*, invocará el procedimiento *create–shadow{}* (espacio OTcl) definido dentro de la función *bind(),* el cual llamará a su correspondiente función *create_shadow()* (espacio C++) de la clase *TclClass*.

La función *create_shadow()* invocará a la función *create()* de la clase *ProtocoloClass,* para crear un nuevo objeto compilado, posteriormente se invocará a los respectivos constructores de sus clases base para inicializar las variables y a su vez reflejarlas en el espacio OTcl, finalizando así el proceso de vinculación entre los dos espacios.

En la Figura 3.60 se presentan las jerarquías para el nuevo objeto de red, en donde se puede observar que no existe reflejo en el espacio OTcl de las clases *Connector* y *NsObject*.

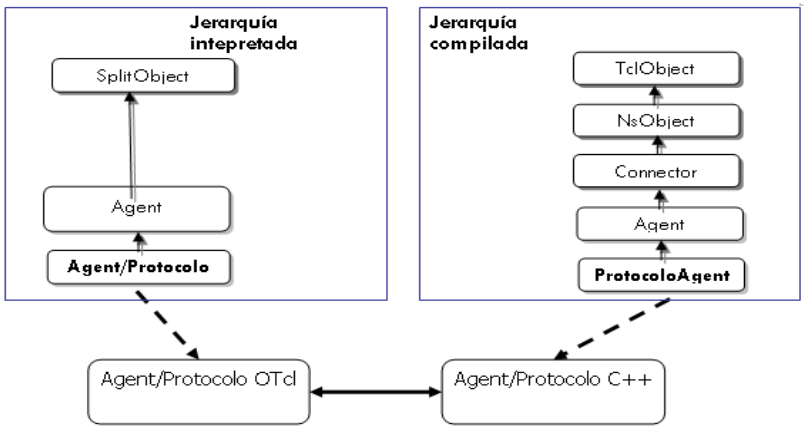


Figura 3.60 Jerarquía interpretada y compilada **[18]**.

Además, la función *create_shadow()* inserta una referencia del nuevo objeto compilado en la tabla *hash*[1]*,* y crea un procedimiento *cmd{}* (explicado en la Sección 1.3.3.2.2.) para su correspondiente objeto interpretado.

Con el objeto interpretado se podrá invocar a procedimientos OTcl o a su vez invocar a funciones en C++ (explicado en la Sección 1.3.3.2.2) para iniciar con la programación de eventos.

Una vez que ocurra el evento planificado, el *kernel* del simulador invocará a la función *recv()* del objeto de red que debe procesarlo, en este caso al de la clase *ProtocoloAgent*.

[1] Tabla interna del intérprete, almacena una referencia para cada objeto *TclObject* de la jerarquía compilada.

1.9.2. PROCESO DE VINCULACIÓN C++/OTCL DE CABECERAS

Como se mencionó anteriormente, la unidad fundamental para el intercambio de información es un objeto de la clase *Packet*, el mismo que está compuesto de un conjunto de cabeceras que representan las unidades de datos para los diferentes protocolos.

Cada cabecera implementada en el espacio C++ tiene su correspondencia en el espacio OTcl, esto se consigue implementando una clase derivada de la clase *PacketHeaderClass* que a su vez es derivada de la clase *TclClas,* que como se indicó anteriormente, provee funcionalidad para el proceso de vinculación.

Para describir el proceso de vinculación se ha tomado como ejemplo la estructura que representa a una *frame* cuyo nombre en el espacio C++ es *hrd_frame* y su correspondiente en el espacio OTcl es *PacketHeader/Frame* (ver la Figura 3.61).

Definición de la estructura que representa a la unidad de datos

```
/*Definición de la estructura hrd_frame*/
typedef struct hdr_frame{

// Variables que representan los campos de la frame
      int seq_var;          // Número de secuencia
      int ack_var;          // Número de confirmación

      static int offset_;   // Variable que permite
                            // acceder a la frame

// Función que permite acceder a la frame
      inline static hdr_frame* access(Packet* p)
      { return (hdr_frame*) p->access(offset_);
      }
}frame;// Nombre con el que se identifica a la
       // estructura hrd_frame
```

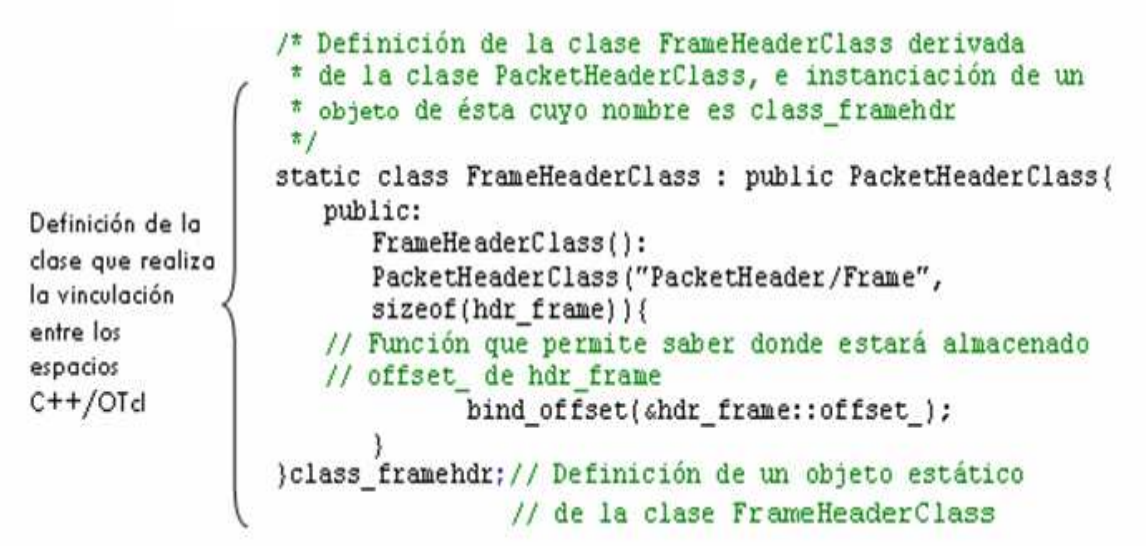


Figura 3.61 Implementación de una unidad de datos en el espacio C++.

De igual forma que en los objetos de red, el proceso de vinculación empieza cuando se ejecuta el simulador NS-2, en donde se inicializan los objetos estáticos, haciendo referencia a la Figura 3.61, se inicializará un objeto *class_framehdr*, lo que implica que se llamará al constructor de la clase *FrameHeaderClass*, éste invocará a los constructores de sus clases base es decir *PacketHeaderClass y TclClass* enviándoles como argumento el nombre de la clase correspondiente en el espacio OTcl *(PacketHeader/Frame);* el proceso de vinculación que se desencadena será similar al descrito para los objetos de red.

Adicionalmente, el constructor *FrameHeaderClass* recibe como argumento el tamaño de la estructura *hdr_frame* calculado en tiempo de compilación; además se invoca a la función *bind_offset()* de la clase *PacketHeaderClass,* enviando como argumento la dirección relativa a la cabecera implementada.

1.9.3. DIAGRAMA DE SECUENCIA

A continuación en la Figura 3.62 y Figura 3.63 se presentan los diagramas de secuencia entre los objetos que participan en el proceso de vinculación entre los espacios C++ y OTcl. Los diagramas se han basado en la descripción realizada en la Sección 3.3.1.

Creación de la Jerarquía Interpretada

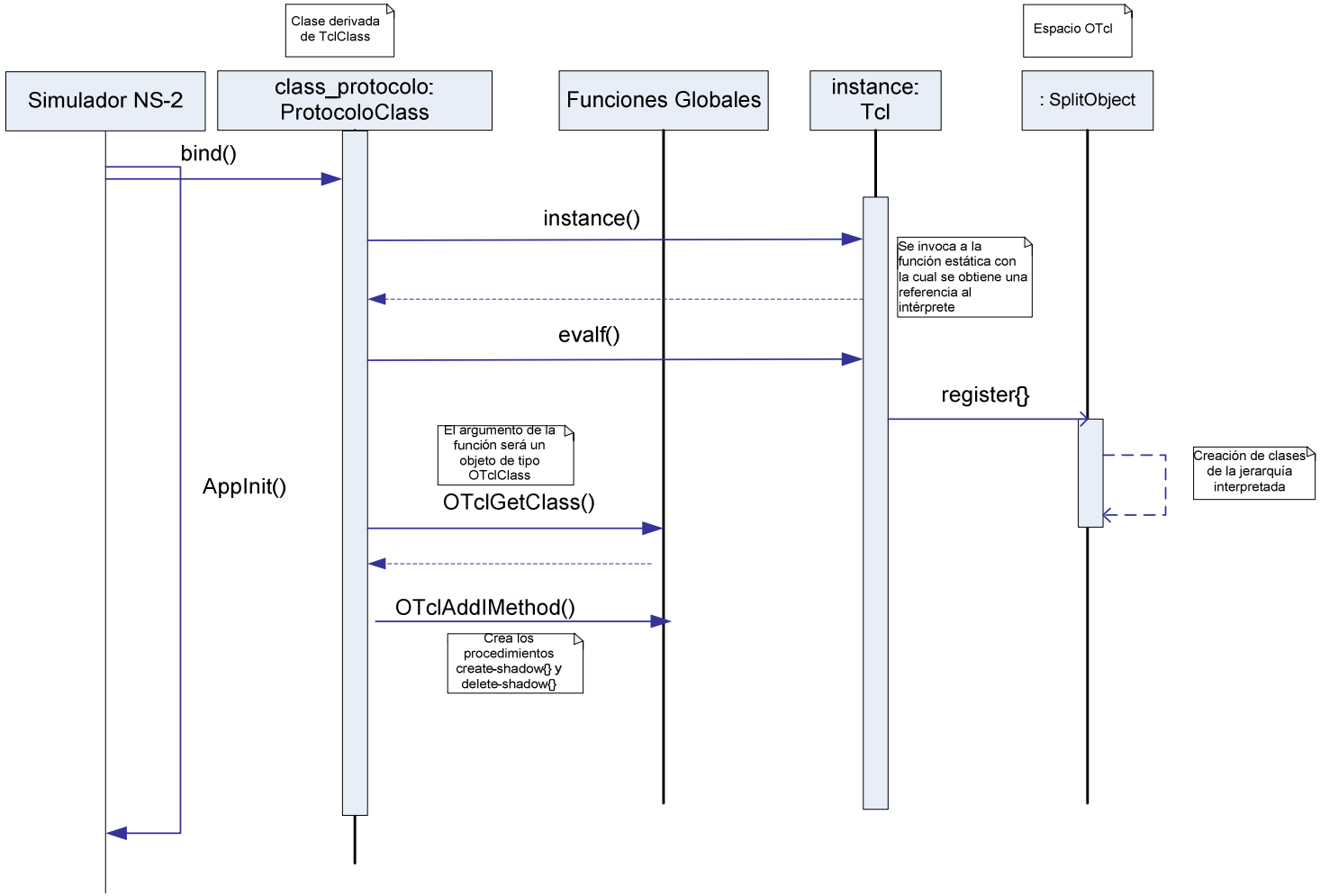


Figura 3.62 Creación de la Jerarquía Interpretada.

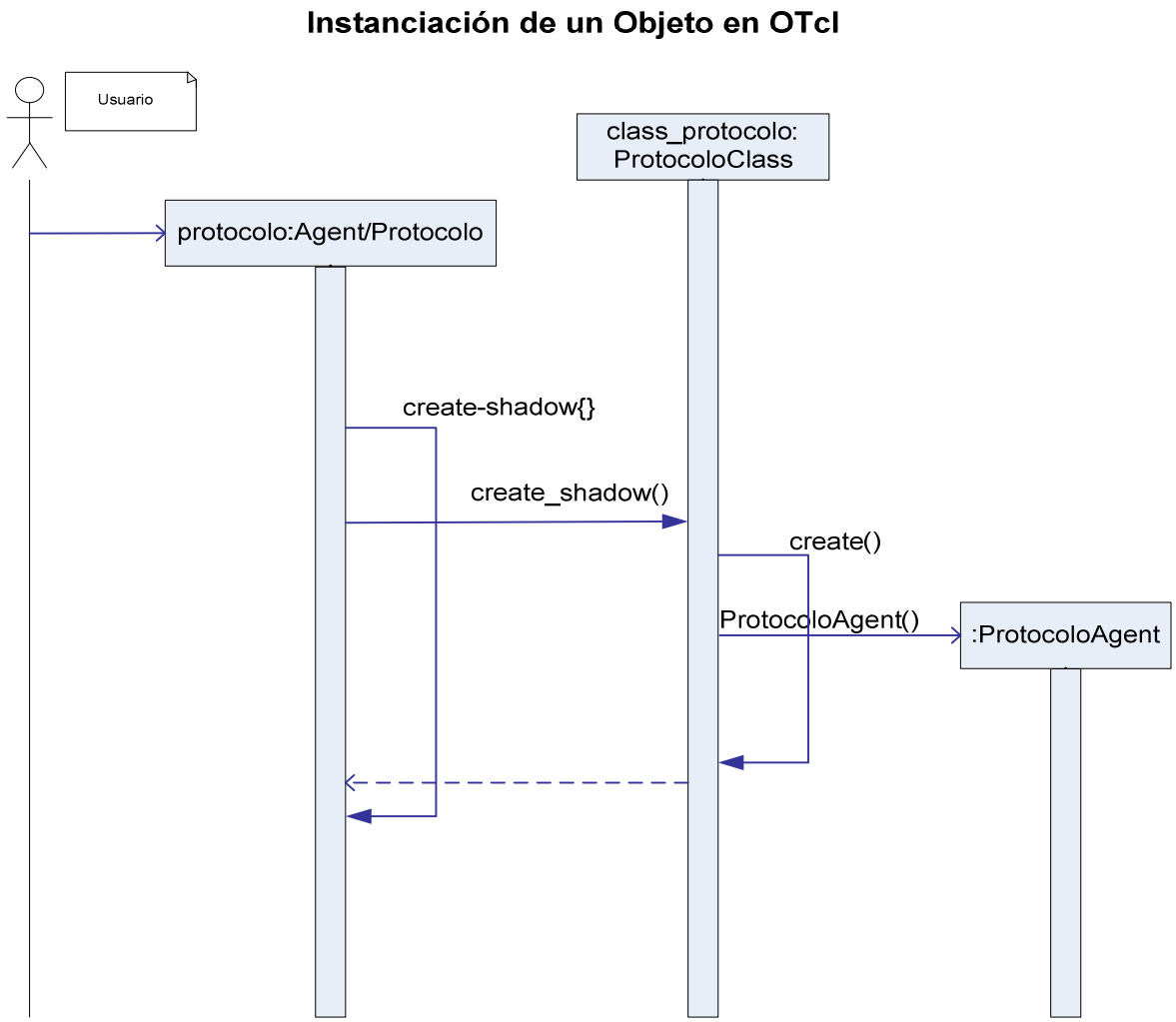


Figura 3.63 Instanciación de un objeto en el espacio OTcl

CAPÍTULO 4

CONCLUSIONES Y RECOMENDACIONES

4.1 CONCLUSIONES

- De la experiencia adquirida en la implementación de los mismos protocolos en los dos simuladores y comparando entre ellos, se puede concluir que en ambos simuladores se separa la implementación de la funcionalidad de un protocolo, de la configuración del escenario de simulación a través del uso de lenguajes diferentes; es así, que para la implementación de la funcionalidad en ambos simuladores se utiliza el lenguaje C++, mientras que para la configuración del escenario de simulación en OMNET++ se utiliza el lenguaje NED y en NS-2 se utiliza el lenguaje OTcl.
- Luego de haber implementado nuevos elementos para ambos simuladores, queda en evidencia las ventajas de la orientación a objetos, ya que facilita la reutilización de código y además a través de ésta se puede identificar y comprender claramente los componentes y su funcionalidad, por ejemplo en las simulaciones realizadas tanto para OMNET++ como para NS-2 se puede diferenciar la implementación de un canal de comunicación como un componente distinto de la implementación de un protocolo que en este caso es un nuevo componente dentro del simulador.

- De la experiencia adquirida y, comparando las facilidades que ofrece cada uno de los simuladores en cuanto a la implementación de la unidad de datos, se pude afirmar que el simulador OMNET++ presenta mayores facilidades, ya que provee el compilador opp_msg, el cual a partir de la definición de la trama, generó la definición e implementación de la clase que la representa, de esta manera se optimizó el tiempo de la implementación.
- Algunos de los protocolos del libro “Redes de Computadores” hicieron uso de temporizadores, los cuales fueron implementados de manera diferente en cada uno de los simuladores; es así que en el caso de OMNET++ para la implementación del temporizador fue necesaria la utilización de las funciones provistas por la clase *cSimpleModule,* mientras que en NS-2 fue necesario implementar una clase que represente al temporizador e instanciar un objeto de ésta en el respectivo protocolo, concluyendo de ésta manera que OMNET++ presenta mayor facilidad en cuanto a la implementación de temporizadores.
- En algunos de los protocolos del libro “Redes de Computadoras” se menciona la posibilidad de que el canal introduzca errores o se produzcan pérdidas, es así que, en el simulador NS-2 se utilizó las clases que permiten modelar errores y pérdidas de unidades de datos en tanto que en el simulador OMNET++ fue necesario añadir funcionalidad que permita cumplir con este objetivo, concluyendo así que el simulador NS-2 provee mayor facilidad para modelar canales de comunicación que se acerquen a la realidad.
- Una vez terminada la implementación de los protocolos se procedió a realizar las respectivas simulaciones en los dos simuladores con lo que se pudo observar que los resultados obtenidos para los protocolos “simplex sin restricciones” y “simplex de parada y espera” fueron idénticos debido a que la configuración de los parámetros de los escenarios de simulación fueron iguales, mientras que los resultados obtenidos de la simulación de los protocolos sobre canales con ruido fueron levemente diferentes en cuanto a los tiempos en que ocurrieron los eventos, esto se debió a que la

generación de errores o ruido dependieron de valores aleatorios, sin embargo cabe recalcar que los resultados están de acuerdo a lo que se espera de la funcionalidad de los protocolos.

- Con el estudio de los respectivos simuladores se puede observar que tanto OMNET++ como NS-2 ofrecen una interfaz gráfica que permite una visualización de la ejecución del escenario de simulación configurado, en nuestra opinión, la interfaz que ofrece el simulador OMNET++ es amigable al usuario ya que a la vez permite observar la topología de la red y los eventos que en ella ocurren, adicionalmente, se puede observar el momento actual de la simulación y los mensajes de depuración que se hayan programado lo que le permite al usuario un mejor control del escenario; a diferencia de éste el simulador NS-2 ofrece una interfaz gráfica en la cual se puede observar la topología de la red y los eventos que ocurren en ella, también se puede observar los mensajes de depuración programados, pero a diferencia de OMNET++, esto ocurre de manera independiente por lo que dificulta al usuario relacionar los resultados presentados en una u otra forma.

- En el caso particular de la implementación de los protocolos de este proyecto, en nuestra apreciación se puede enumerar algunos de los aspectos por los cuales el desarrollo en OMNET++ es más fácil que en NS-2, entre ellos están: el hecho de que se trabajó sobre el ambiente Windows y además se utilizó el software Visual Studio.NET lo cual facilitó la compilación, depuración que facilitaron el desarrollo; en tanto que NS-2 se trabajó sobre el ambiente Linux y no se utilizó un software que facilite la depuración; otro aspecto que facilitó la implementación es el hecho de que la funcionalidad requerida en la implementación de los nuevos componentes se concentran en muy pocas clases, lo que facilita el entendimiento para el desarrollo, en tanto que, la funcionalidad ofrecida en NS-2 se concentra en un gran número de clases, lo que implica que el entendimiento sea un tanto complejo.

- La descripción realizada para la interacción del simulador OMNET++ con un nuevo protocolo fue un tanto compleja ya que no existió información en

cuanto a este proceso por lo que, nos basamos en el código fuente provisto por el simulador; en tanto que para el proceso de interacción del simulador NS-2 con un nuevo protocolo nos basamos en la información provista en el manual “Ns Notes and Documentation” del simulador.

4.2 RECOMENDACIONES

- Se recomienda que el usuario tenga una clara visión de las clases que proveen los simuladores y de su funcionalidad, para de esta manera, reutilizarlas de acuerdo a los requerimientos de las nuevas implementaciones**.**
- Debido a que la transmisión de tramas con errores en el simulador NS-2 no se las pueden apreciar en la aplicación NAM, se sugiere al usuario revisar el archivo de traza con extensión .tr, ya que en éste las tramas con error se las identifica de acuerdo al valor registrado en el campo de banderas.
- Se recomienda configurar valores adecuados para los temporizadores, éstos valores deben estar acorde a los valores configurados para los parámetros del escenario de simulación tales como: velocidad de transmisión, tiempo de propagación en el canal y tamaño de la trama.
- Se sugiere tener cierto conocimiento de los lenguajes utilizados para la configuración de los escenarios de simulación tanto para el simulador OMNET++ como para NS-2.
- Tomar en cuenta que los cambios que se realicen en lenguaje C++ tanto para el simulador OMNET++ como para NS-2 necesariamente se debe realizar una re-compilación para registrar estos cambios; esto no es necesario si se hacen cambios en cuanto a la configuración de parámetros sean en el archivo omnetpp.ini o en el script Tcl.
- Este proyecto puede ser aplicado para cumplir dos objetivos: uno orientado al área académica y otro, a la motivación para desarrollar nuevas implementaciones de componentes en estos simuladores. Desde el punto de vista académico, este trabajo puede ser utilizado como herramienta de ayuda en la materia de Redes LAN que es impartida en la carrera de Electrónica y Redes de Información.

BIBLIOGRAFÍA

[1] A. Varga, "OMNeT++ User Manual", http://www.omnetpp.org, 2005.

[2] K. Fall, K. Varadhan, "Ns Notes and Documentation", The VINT Project. UC Berkeley, LBN, http://www.isi.edu/nsnam/ns/ns_doc, 2006.

[3] P. Gonzáles, “Modelado Y Simulación de RHTP: Protocolo de encaminamiento de Árbol Jerárquico”, Universidad de Alcalá, htttp://www.it.aut.uah.es/gibanez/ProyectoPedro.pdf, ,2007.

[4] S.N. “Herramientas Software Para La Simulación De Redes De Comunicaciones”, http://www.nsl.csie.nctu.edu.tw/NCTUnsReferences/

capitulo4.pdf

[5] Eitan Altman, Tatiana Jiménez, "NS Simulator for beginners", http://www-sop.inria.fr/maestro/personnel/Eitan.Altman/COURS-NS/n3.pdf, 2004.

[6] P, Anelli & E. Horlait, ”NS-2: Principes de conception et d'utilisation”,http:// www-sop.inria.fr/rodeo/personnel/Pierre.Ansel/Manuel_NS1.3.pdf.

[7] Andrew S.Tanenbaum, *Redes de Computadoras*, TERCERA EDICIÓN, PRENTICE. Pag:184-228.

[8] A. Varga, "OMNeT++ API", http://www.omnetpp.org/doc/api/index.html, 2005.

[9] A. Varga, Ahmet Sekercioglu "OMNeT++ Tutorial", http: //www.omnetpp. .org /doc/tictoc-tutorial/, 2005

[10] “Network Simulator ns-2”, http://www.cse.ohio-state.edu/~prasun/788/ ns2.ppt

[11] J. Chung & M. Claypool, “ NS by Example”, http://nile.wpi.edu/NS /NS_ by_Example.pdf

[12] Ibán Cereijo, “Simulación y comparativa de mecanismos de conmutación en redes ópticas”,Universidad de Vigo, http://www-gris.det.uvigo.Es/~ mveiga/ optinet6 /doc/SimOBS.pdf -

[13] Matthias Transier, “ns-2 Tutorial, Extending the Simulator”, Universidad Mannheim, http://www.tm.uka.de/forschung/SPP1140/events/meetings/ns2 /extending.pdf.gz -

[14] Francisco J. Ros & Pedro M. Ruiz, “Implementing a New Manet Unicast Routing Protocol in NS2”, Universidad de Mursia, http:// www.masimum.dif.um.es/nsrt-howto/pdf/nsrt-howto.pdf, 2004.

[15] Dr. Mark J. Sebern, “STL Vector Clase”, http://www.msoe.edu/eecs/ce/ courseinfo /stl /vector.htm, 1998.

[16] C++ STL Tutorial and Examples,http://www.yolinux.com/TUTORIALS/ LinuxTutorialC++STL.html

[17] Changjee Joo,NS-2 : "The network simulator", Network Lab., SNU , http://cnslab.snu.ac.kr/board/2005_NS2.pdf, 2005.

[18] Jae-Pil Yoo,”How to use NS2”, Konkuk university, http:// J1-1À¯ ÀçÇ[1].pdf

ANEXO A

INSTALACIÓN DEL SIMULADOR OMNET++ SOBRE LA PLATAFORMA WINDOWS XP

El simulador OMNET++ puede funcionar tanto en Windows como en Linux. El principal requerimiento para su funcionamiento es tener un compilador C++.

DESCARGA DEL PROGRAMA

La fuente principal en donde se puede obtener el instalador del simulador OMNET++ es en el sitio web oficial; el instalador del simulador consiste en el archivo ejecutable omnetpp-3.3-win32.exe.

Esta versión contiene mejoras al interfaz grafico y corrección de errores en versiones anteriores.

Se presentará la instalación de la versión omnetpp-3.3-win32.

COMPONENTES

Los componentes requeridos y opcionales para el funcionamiento del simulador OMNET++, se presentan a continuación en la Figura A1.

Componente	Requerido	Opcional
Tcl/Tk	X	
BLT 2.4z	X	
bison/flex		X
Perl	X	
iconv	X	
libxml, libxslt	X	

Figura A1. Componentes para el simulador OMNET++**.**

En el sitio web oficial mencionado se podrá encontrar diferentes versiones del software que se ajustará según el requerimiento del usuario.

INSTALACIÓN

Una vez descargado el instalador del simulador OMNET++ (omnetpp-3.3-win32) proceder a su instalación. A continuación se detallan los pasos para la instalación en el sistema operativo Windows (Windows XP Profesional Version 2002).

1. Una vez que se ejecute el instalador del simulador aparecerá un cuadro de diálogo que ayudará al usuario en el proceso.
2. En el cuadro de diálogo, el usuario deberá escoger los componentes que desee instalar. Se recomienda instalar los componentes señalados por defecto. (ver la Figura A2.)

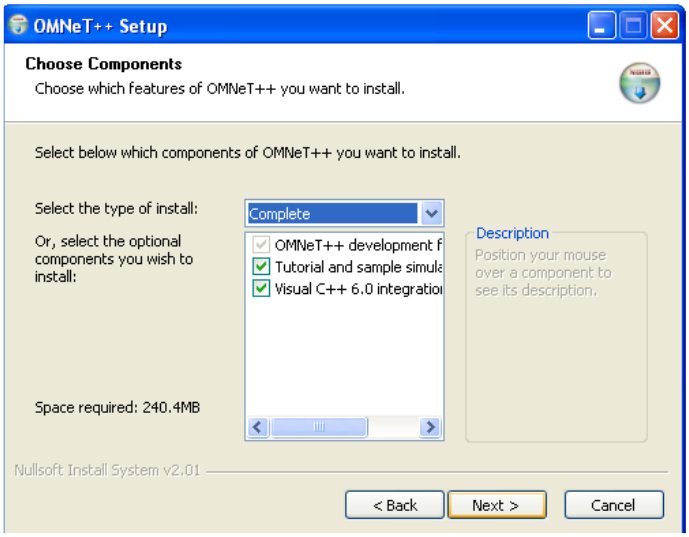


Figura A2. Cuadro de diálogo para escoger los componentes.

3. Al seguir en el proceso de instalación aparecerá un cuadro de diálogo en el cual se debe indicar la ubicación de donde está instalado el programa *Ghostscript*, pero esto es opcional por lo que se puede omitir este paso y proseguir. (ver la Figura A3.)

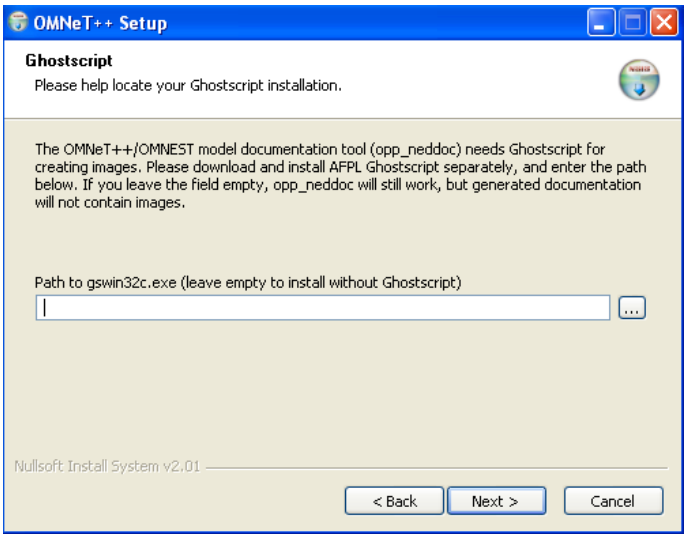


Figura A3. Cuadro de diálogo para la ubicación del programa *Ghostscript*.

4. En el siguiente cuadro de diálogo el usuario deberá escoger las librerías que va a utilizar de acuerdo a la versión de Visual Studio que tenga instalado, para este caso al tener instalado el Visual Studio .NET 2003 se ha escogido v71-release.(ver la Figura A4.)

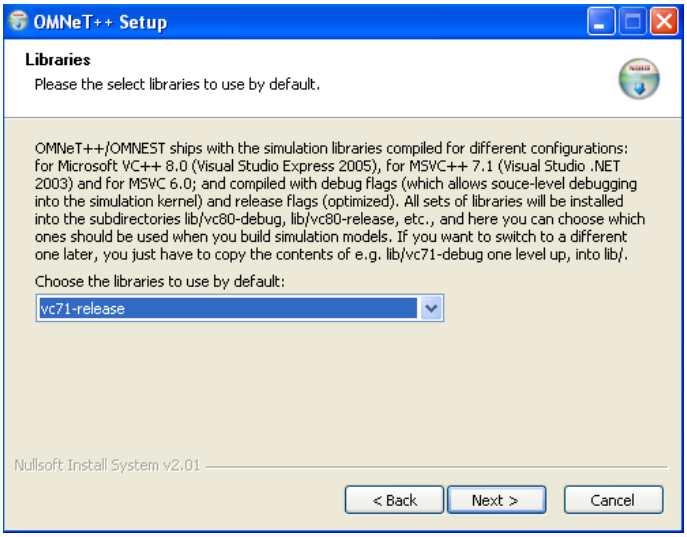


Figura A4. Cuadro de diálogo para escoger la versión de librerías del simulador.

5. Con los pasos anteriores realizados se finaliza la instalación.

ESTRUCTURA DE LOS DIRECTORIOS EN OMNET++

Una vez instalado el programa de simulación, se puede obtener el directorio y los subdirectorios siguientes como se muestra en la Figura A5.

- **OMNeT**++: Directorio raíz.
- **bin:** Este directorio contiene los ejecutables *nedtool*, *gned* y otros.
- **include:** Este directorio contiene los archivos de cabecera para los modelos de simulación.
- **lib:** Este directorio contiene archivos de biblioteca
- **bitmaps:** Este directorio contiene los iconos que pueden ser usados en el diseño de la red.
- **doc:** Este directorio contiene algunos manuales en formato PDF, readme, etc.
- **src:** Este directorio contiene los códigos fuente para los componentes del simulador.

Sub-directorios de src

- **nedc:** Este directorio contiene los compiladores *nedtool*, *oppmsgc*.
- **sim:** Este directorio contiene el código fuente que conforma el *kernel* del simulador.
- **parsim:** Este directorio contiene archivos para ejecuciones distribuidas (subdirectorio de *sim*).
- **netbuilder:** Este directorio contiene archivos para la lectura dinámica de archivos NED (subdirectorio de *sim*).
- **envir:** Este directorio contiene código común para las interfaces de usuario.

...

Los códigos fuente de ejemplos de simulación se encuentran en el directorio **samples,** dentro de éste se puede encontrar por ejemplo:

- **aloha:** Este directorio contiene el modelo para el protocolo Aloha.
- **p-to-p:** Este directorio contiene el modelo para el protocolo Peer to Peer

...

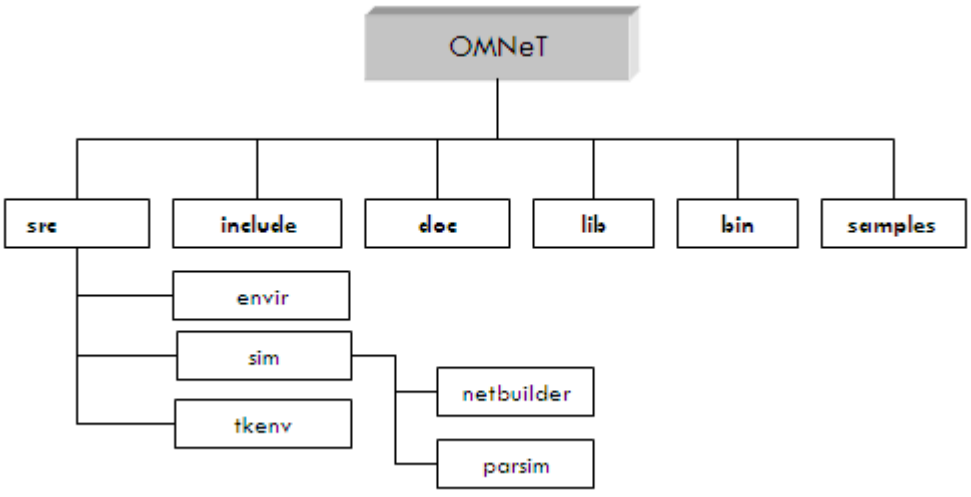


Figura A5. Parte de los directorios del simulador OMNet++.

UBICACIÓN DE LAS CLASES Y ARCHIVOS

CLASES	ARCHIVO	UBICACIÓN
cSimulation	csimul.h, csimul.cc	OMNeT\src\include OMNeT++\src\sim
cEnvir	cenvir.h, cenvir.cc	OMNeT\src\include OMNeT++\src\envir
cModuleType	ctypes.h, ctypes.cc	OMNeT\src\include OMNeT++\src\envir
cModuleInterface	ctypes.h, ctypes.cc	OMNeT\src\include OMNeT++\src\envir
cNetworkType	ctypes.h, ctypes.cc	OMNeT\src\include OMNeT++\src\envir
cSimpleModule	csimplemodule.h; csimplemodule.cc	OMNeT\src\include OMNeT++\src\sim
cMessage	cmessage.h, cmessage.cc	OMNeT\src\include OMNeT++\src\sim
-	globals.h, globals.cc	OMNeT\src\include OMNeT++\src\sim
cArray	carray.h; carray.cc	OMNeT\src\include OMNeT++\src\sim
cPar	cpar.h; cpar.cc	OMNeT\src\include OMNeT++\src\sim

-	macros. H	OMNeT\src\include

Figura A6. Ubicación de clases y archivos del simulador OMNET++.

ANEXO B

INSTALACIÓN DEL SIMULADOR NS-2 SOBRE LA PLATAFORMA LINUX

El simulador NS-2 puede funcionar tanto en Windows como en Linux. El principal requerimiento para su funcionamiento es tener un compilador C++.

DESCARGA DEL PROGRAMA

La fuente principal en donde se puede obtener el instalador del simulador NS-2 es en el sitio web oficial en donde se lo puede encontrar en diferentes formas:

- **ns-src-2.30.tar.gz:** Contiene el código fuente del simulador.
- **ns-allinone-2.30.tar.gz**: Además de contener el código fuente, contiene todas las librerías y programas necesarios para la compilación y uso del simulador NS-2(como por ejemplo OTcl).

En el sitio web antes mencionado se recomienda instalar la versión del programa por paquetes a aquellos usuarios que desean ahorrar espacio de memoria.

Se presentará la instalación de la versión ns-2.30 (ns-allinone-2.30.tar.gz).

COMPONENTES

El paquete ns-allinone-2.30 contiene componentes requeridos y opcionales para el funcionamiento del simulador NS-2, a continuación en la Figura B1 se presentan los componentes.

	Componente	Requerido	Opcional
Tcl	*tcl release 8.4.13*	**X**	
Tk	*tk release 8.4.13*	**X**	
OTcl	*otcl release 1.12*	**X**	
TclCl	*tclcl release 1.18*	**X**	
NS	*ns release 2.30*	**X**	
Nam	*Nam release 1.12*		**X**
Xgraph	*Xgraph version 12*		**X**
GT-ITM	*Georgia Tech Internetwork Topology Modeler*		**X**
SBG	*Stanford GraphBase package*		**X**
CWeb	*CWeb version 1.0 g*		**X**
ZLib	*zlib version 1.2.3*		**X**

Figura B1. Componentes del paquete ns-allinone-2.30**.**

INSTALACIÓN

Una vez descargado el paquete completo del instalador de NS-2 (ns-allinone-2.30.tar.gz), se puede proceder a su instalación. A continuación se detallan los pasos necesarios para instalar en un sistema operativo Linux (Fedora Core versión 2.6.9).

1. Descomprimir el fichero fuente en el directorio en el que se desea instalar NS-2, para lo cual se utiliza las herramientas **tar** y **gzip.** Por ejemplo se instalará en el directorio Programas, en donde se ejecuta el comando que se presenta en la Figura B2.

```
[root@localhost Programas]#tar zxvf ns-allinone-2.30.tar.gz
```

Figura B2. Descompresión del fichero fuente.

2. Una vez descomprimido el directorio, se procederá a instalar la herramienta mediante el *script* denominado *install*[1]**,** el cual automáticamente compilará e instalará todo el paquete (ver la Figura B3).

```
[root@localhost Programas]#cd ns-allinone-2.30

[root@localhost ns-allinone-2.30]#./install
```

Figura B3. Ejecución del *script install*.

3. Posteriormente para que NS-2 funcione correctamente, se debe configurar el entorno del *shell*, para lo cual se debe insertar en el archivo *profile* (ubicado en el directorio /etc/) la siguiente sección de código que se muestra en la Figura B4.

```
export
PATH=/Programas/ns-allinone-2.30/bin:/Programas/ns-allinone-
2.30/tcl8.4.13/unix:/Programas/ns-allinone-2.30/tk8.4.13/unix:$PATH

export
LD_LIBRARY_PATH=/Programas/ns-allinone-2.30/otcl-
1.12:/Programas/ns-allinone-2.30/lib:/Programas/ns-allinone-
2.30/tcl8.4.13/unix:/Programas/ns-allinone-
2.30/tk8.4.13/unix:$LD_LIBRARY_PATH

export
TCL_LIBRARY=/Programas/ns-allinone-2.30/tcl8.4.13/library
```

Figura B4. Sección de código en el archivo *profile* para la configuración del entorno del *shell.*

4. Finalmente es conveniente hacer una verificación de que la instalación se efectuado correctamente, para esto se debe ejecutar el *script validate* localizado en el directorio /ns-allinone-2.30/ns-2.30/, con la instrucción que se presenta en la Figura B5**.**

```
[root@localhost ns-allinone-2.30]# cd ns-2.30/
[root@localhost ns-2.30]#./validate
```

Figura B5. Verificación de la correcta instalación a través de la ejecución del *script validate*.

Si no aparece ningún mensaje de error luego de haber ejecutado el comando anterior, se puede dar a la instalación como exitosa.

[1] Script diseñado para ejecutar los archivos configure y *make* desde el directorio ns-2.30, luego de ello también hace lo mismo desde el directorio tclcl-1.18.

ESTRUCTURA DE LOS DIRECTORIOS EN NS-2

Los principales directorios del paquete ns-allinone-2.30 son:

- **ns-2.30 :** Este directorio contiene el código fuente del simulador, aquí se puede localizar los directorios que contienen la implementación de los objetos de red desarrollados.

Las nuevas implementaciones deben ser ubicadas dentro de éste directorio.

En éste también se puede localizar el archivo ***Makefile*** en el cual se indica los archivos que se van a compilar para crear los archivos **.o**.

- **tclcl-1.18:** En éste se encuentra los archivos cuyo código constituye el interfaz con el intérprete.
- **nam-1.12:** Este contiene la implementación de la interfaz gráfica NAM para el simulador.

Sub-directorios de ns-2.30

- **common**: Este contiene el código fuente C++ de las clases que sirven como base para la implementación de componentes de red básicos.
- **tcl:** En éste se encuentran los siguientes sub-directorios:
- ***lib:*** Este contiene el código fuente OTcl para muchas partes básicas y esenciales de la implementación (agentes, nodos, enlaces, paquetes, direcciones, ruteo, etc.),
- ***ex:*** Este contiene ejemplos de configuración en código OTcl.

En la Figura B6. se presenta parte de los directorios que se originan al instalar el paquete ns-allinone-2.30.

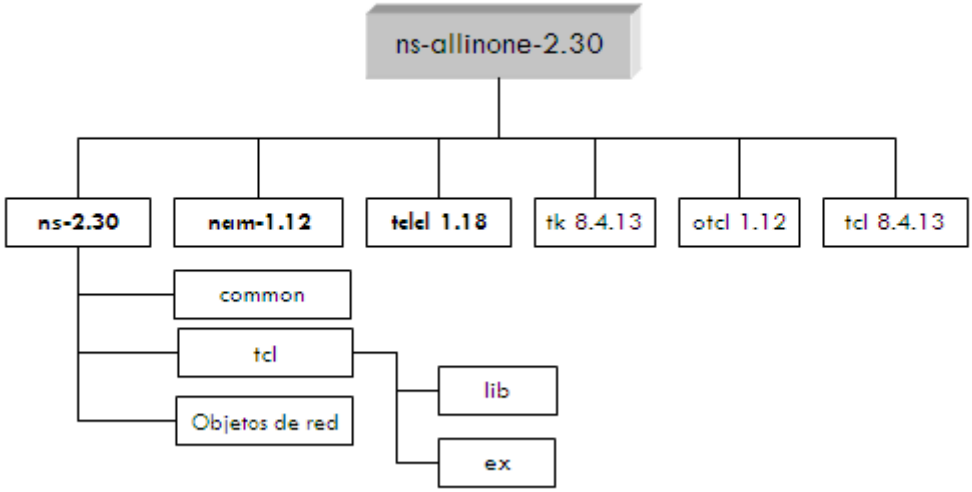


Figura B6. Parte de los directorios del paquete ns-allinone-2.30.

UBICACIÓN DE LAS CLASES Y ARCHIVOS

CLASES	ARCHIVO	UBICACIÓN
TclObject	tclcl.h; tcl.cc	ns-allinone-2.30\tclcl-1.18
Tcl	tclcl.h; tcl.cc	ns-allinone-2.30\tclcl-1.18
TclClass	tclcl.h; tcl.cc	ns-allinone-2.30\tclcl-1.18
SplitObject	tcl-object.tcl	ns-allinone-2.30\tclcl-1.18
NsObject	object.h; object.tcl	ns-allinone-2.30\ns-2.30\common

Connector	connector.h; connecto.cc	ns-allinone-2.30\ns-2.30\common
Agent	agent.h; agent.cc	ns-allinone-2.30\ns-2.30\common
Packet	packet.h; packet. cc	ns-allinone-2.30\ns-2.30\common
*******	Ns-default.tcl	ns-allinone-2.30\ns-2.30\tcl\lib
Classifier	classifier.h; classifier.cc	ns-allinone-2.30\ns-2.30\common

Figura B7. Componentes del paquete ns-allinone-2.30**.**

ANEXO C

COPIAS LIBRO “REDES DE COMPUTADORAS”

ANDREW TANENBAUM

3

LA CAPA DE ENLACE DE DATOS

En este capítulo estudiaremos los principios de diseño de la capa 2, la capa de enlace de datos. Este estudio tiene que ver con los algoritmos para lograr una comunicación confiable y eficiente entre dos máquinas adyacentes en la capa de enlace de datos. Por adyacente, queremos decir que las dos máquinas están conectadas por un canal de comunicaciones que actúa de manera conceptual como un alambre (por ejemplo, un cable coaxial, una línea telefónica o un canal inalámbrico de punto a punto). La propiedad esencial de un canal que lo hace asemejarse a un alambre es que los bits se entregan con exactitud en el mismo orden en que fueron enviados.

A primera vista podría pensarse que este problema es tan trivial que no hay ningún software que estudiar: la máquina *A* sólo pone los bits en el alambre, y la máquina *B* simplemente los toma. Por desgracia, los circuitos de comunicación cometen errores ocasionales. Además, tienen una tasa de datos finita y hay un retardo de propagación diferente de cero entre el momento en que se envía un bit y el momento en que se recibe. Estas limitaciones tienen implicaciones importantes para la eficiencia de la transferencia de datos. Los protocolos usados para comunicaciones deben considerar todos estos factores. Dichos protocolos son el tema de este capítulo.

Tras una introducción a los aspectos clave de diseño presentes en la capa de enlace de datos, comenzaremos nuestro estudio de sus protocolos observando la naturaleza de los errores, sus causas y la manera en que se pueden detectar y corregir. Después estudiaremos una serie de protocolos de complejidad creciente, cada uno de los cuales resuelve los problemas presentes en esta capa. Por último, concluiremos con un estudio del modelado y la corrección de los protocolos y daremos algunos ejemplos de protocolos de enlace de datos.

3.1 CUESTIONES DE DISEÑO DE LA CAPA DE ENLACE DE DATOS

La capa de enlace de datos tiene que desempeñar varias funciones específicas, entre las que se incluyen:

1. Proporcionar una interfaz de servicio bien definida con la capa de red.
2. Manejar los errores de transmisión.
3. Regular el flujo de datos para que receptores lentos no sean saturados por emisores rápidos.

Para cumplir con estas metas, la capa de enlace de datos toma de la capa de red los paquetes y los encapsula en **tramas** para transmitirlos. Cada trama contiene un encabezado, un campo de carga útil (*payload*) para almacenar el paquete y un terminador o final, como se ilustra en la figura 3-1. El manejo de las tramas es la tarea primordial de la capa de enlace de datos. En las siguientes secciones examinaremos en detalle todos los aspectos mencionados.

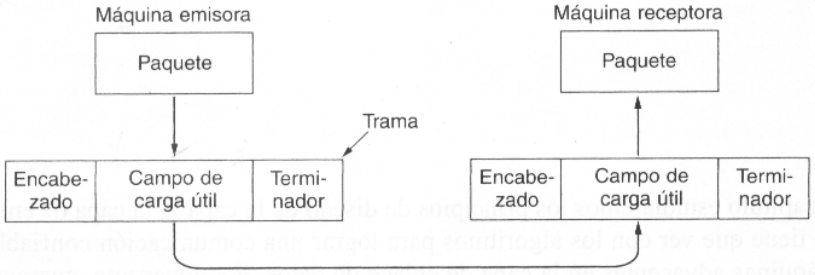


Figura 3-1. Relación entre los paquetes y las tramas.

Aunque este capítulo sólo analiza la capa de enlace de datos y los protocolos de enlace de datos, muchos de los principios que analizaremos aquí, como el control de errores y el de flujo, también se encuentran en la capa de transporte y en otros protocolos. De hecho, en muchas redes, estas funciones se encuentran sólo en las capas superiores y no en la de enlace de datos. Sin embargo, independientemente de donde se encuentren, los principios son casi los mismos, por lo que en realidad no importa en qué parte del libro los analicemos. Por lo general, éstos se muestran en la capa de enlace de datos en sus formas más simples y puras, por lo que dicha capa es un buen lugar para examinarlos en detalle.

3.1.1 Servicios proporcionados a la capa de red

La función de la capa de enlace de datos es suministrar servicios a la capa de red. El servicio principal es transferir datos de la capa de red en la máquina de origen a la capa de red en la máquina de destino. En la capa de red de la máquina de origen hay una entidad, llamada proceso, que entrega algunos bits a la capa de enlace de datos para transmitirlos a la máquina de destino. El tra-

bajo de la capa de enlace de datos es transmitir los bits a la máquina de destino, para que puedan ser entregados a su capa de red, como se muestra en la figura 3-2(a). La transmisión real sigue la trayectoria de la figura 3-2(b), pero es más fácil pensar en términos de dos procesos de capa de enlace de datos que se comunican usando un protocolo de enlace de datos. Por esta razón, a lo largo de este capítulo usaremos de manera implícita el modelo de la figura 3-2(a).

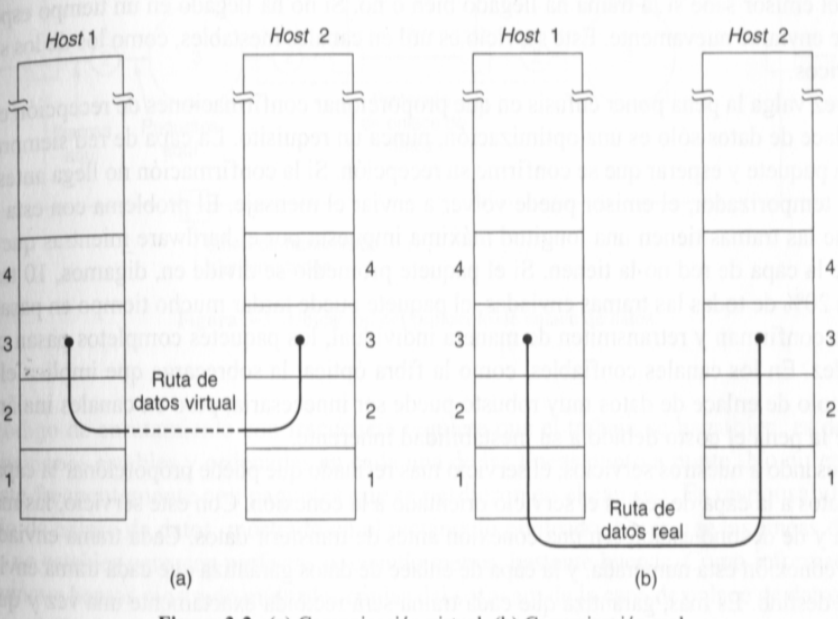


Figura 3-2. (a) Comunicación virtual. (b) Comunicación real.

La capa de enlace de datos puede diseñarse para ofrecer varios servicios. Los servicios reales ofrecidos pueden variar de sistema a sistema. Tres posibilidades razonables que normalmente se proporcionan son:

1. Servicio no orientado a la conexión sin confirmación de recepción.
2. Servicio no orientado a la conexión con confirmación de recepción.
3. Servicio orientado a la conexión con confirmación de recepción.

Consideremos cada uno de ellos por separado.

El servicio no orientado a la conexión sin confirmación de recepción consiste en hacer que la máquina de origen envíe tramas independientes a la máquina de destino sin pedir que ésta confirme la recepción. No se establece conexión de antemano ni se libera después. Si se pierde una trama debido a ruido en la línea, en la capa de enlace de datos no se realiza ningún intento por detectar la pérdida ni por recuperarse de ella. Esta clase de servicio es apropiada cuando la tasa de errores es muy baja, por lo que la recuperación se deja a las capas superiores. También es apropiada para el tráfico en tiempo real, por ejemplo de voz, en el que la llegada retrasada de datos es peor que

los errores de datos. La mayoría de las LANs utilizan servicios no orientados a la conexión sin confirmación de recepción en la capa de enlace de datos.

El siguiente paso hacia adelante en cuanto a confiabilidad es el servicio no orientado a la conexión con confirmación de recepción. Cuando se ofrece este servicio tampoco se utilizan conexiones lógicas, pero se confirma de manera individual la recepción de cada trama enviada. De esta manera, el emisor sabe si la trama ha llegado bien o no. Si no ha llegado en un tiempo especificado, puede enviarse nuevamente. Este servicio es útil en canales inestables, como los de los sistemas inalámbricos.

Tal vez valga la pena poner énfasis en que proporcionar confirmaciones de recepción en la capa de enlace de datos sólo es una optimización, nunca un requisito. La capa de red siempre puede enviar un paquete y esperar que se confirme su recepción. Si la confirmación no llega antes de que expire el temporizador, el emisor puede volver a enviar el mensaje. El problema con esta estrategia es que las tramas tienen una longitud máxima impuesta por el hardware mientras que los paquetes de la capa de red no la tienen. Si el paquete promedio se divide en, digamos, 10 tramas, y se pierde 20% de todas las tramas enviadas, el paquete puede tardar mucho tiempo en pasar. Si las tramas se confirman y retransmiten de manera individual, los paquetes completos pasan con mayor rapidez. En los canales confiables, como la fibra óptica, la sobrecarga que implica el uso de un protocolo de enlace de datos muy robusto puede ser innecesaria, pero en canales inalámbricos bien vale la pena el costo debido a su inestabilidad inherente.

Regresando a nuestros servicios, el servicio más refinado que puede proporcionar la capa de enlace de datos a la capa de red es el servicio orientado a la conexión. Con este servicio, las máquinas de origen y de destino establecen una conexión antes de transferir datos. Cada trama enviada a través de la conexión está numerada, y la capa de enlace de datos garantiza que cada trama enviada llegará a su destino. Es más, garantiza que cada trama será recibida exactamente una vez y que todas las tramas se recibirán en el orden adecuado. En contraste, con el servicio no orientado a la conexión es posible que una confirmación de recepción perdida cause que una trama se envíe varias veces y, por lo tanto, que se reciba varias veces. Por su parte, el servicio orientado a la conexión proporciona a los procesos de la capa de red el equivalente de un flujo de bits confiable.

Cuando se utiliza un servicio orientado a la conexión, las transferencias tienen tres fases distintas. En la primera, la conexión se establece haciendo que ambos lados inicialicen las variables y los contadores necesarios para seguir la pista de las tramas que han sido recibidas y las que no. En la segunda fase se transmiten una o más tramas. En la tercera fase, la conexión se cierra y libera las variables, los búferes y otros recursos utilizados para mantener la conexión.

Considere un ejemplo típico: una subred de WAN que consiste en enrutadores conectados por medio de líneas telefónicas alquiladas de punto a punto. Cuando llega una trama a un enrutador, el hardware la examina para verificar si está libre de errores (mediante una técnica que veremos más adelante en este capítulo), y después la pasa al software de la capa de enlace de datos (que podría estar integrado en un *chip* de la tarjeta de interfaz de red). Dicho software comprueba si ésta es la trama esperada y, de ser así, entrega el paquete contenido en el campo de carga útil al software de enrutamiento. A continuación, este software elige la línea de salida adecuada y reenvía el paquete al software de la capa de enlace de datos, que luego lo transmite. En la figura 3-3 se muestra el flujo a través de dos enrutadores.

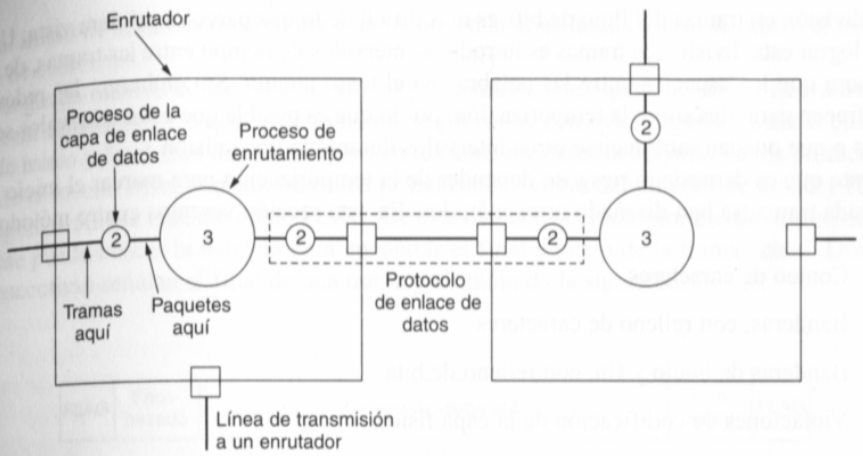


Figura 3-3. Ubicación del protocolo de enlace de datos.

El código de enrutamiento con frecuencia requiere que el trabajo se haga bien, es decir, que haya conexiones estables y ordenadas en cada una de las líneas punto a punto. No quiere que se le moleste frecuentemente con paquetes que se perdieron en el camino. Es responsabilidad del protocolo de enlace de datos, mostrado en el rectángulo punteado, hacer que las líneas de comunicación no estables parezcan perfectas o, cuando menos, bastante buenas. Como información adicional, aunque hemos mostrado múltiples copias del software de la capa de enlace de datos en cada enrutador, de hecho una sola copia maneja todas las líneas, con diferentes tablas y estructuras de datos para cada una.

3.1.2 Entramado

A fin de proporcionar servicios a la capa de red, la de enlace de datos debe utilizar los servicios que la capa física le proporciona. Lo que hace la capa física es aceptar un flujo de bits puros e intentar entregarlo al destino. No se garantiza que este flujo de bits esté libre de errores. La cantidad de bits recibidos puede ser menor, igual o mayor que la cantidad de bits transmitidos, y éstos pueden tener diferentes valores. Es responsabilidad de la capa de enlace de datos detectar y, de ser necesario, corregir los errores.

El método común es que la capa de enlace de datos divida el flujo de bits en tramas separadas y que calcule la suma de verificación de cada trama. (Posteriormente en este capítulo se analizarán los algoritmos de suma de verificación.) Cuando una trama llega al destino, se recalcula la suma de verificación. Si la nueva suma de verificación calculada es distinta de la contenida en la trama, la capa de enlace de datos sabe que ha ocurrido un error y toma medidas para manejarlo (por ejemplo, descartando la trama mala y, posiblemente, regresando un informe de error).

La división en tramas del flujo de bits es más difícil de lo que parece a primera vista. Una manera de lograr esta división en tramas es introducir intervalos de tiempo entre las tramas, de la misma manera que los espacios entre las palabras en el texto común. Sin embargo, las redes pocas veces ofrecen garantías sobre la temporización, por lo que es posible que estos intervalos sean eliminados o que puedan introducirse otros intervalos durante la transmisión.

Puesto que es demasiado riesgoso depender de la temporización para marcar el inicio y el final de cada trama, se han diseñado otros métodos. En esta sección veremos cuatro métodos:

1. Conteo de caracteres.
2. Banderas, con relleno de caracteres.
3. Banderas de inicio y fin, con relleno de bits.
4. Violaciones de codificación de la capa física.

El primer método de entramado se vale de un campo en el encabezado para especificar el número de caracteres en la trama. Cuando la capa de enlace de datos del destino ve la cuenta de caracteres, sabe cuántos caracteres siguen y, por lo tanto, dónde está el fin de la trama. Esta técnica se muestra en la figura 3-4(a) para cuatro tramas de 5, 5, 8 y 8 caracteres de longitud, respectivamente.

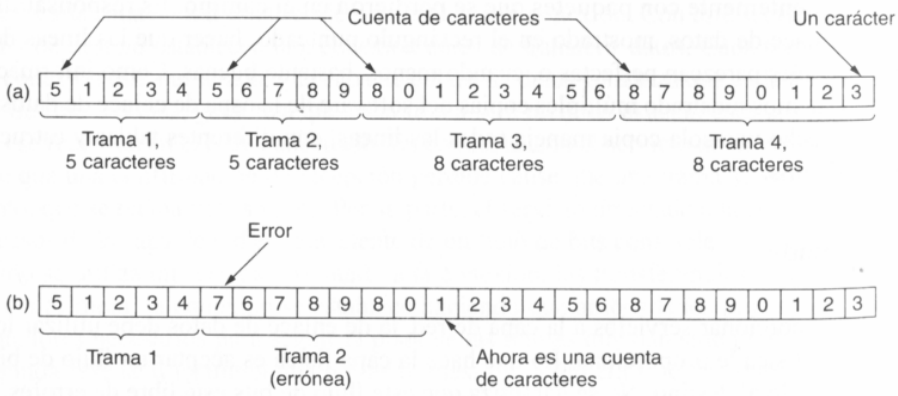


Figura 3-4. Un flujo de caracteres. (a) Sin errores. (b) Con un error.

El problema con este algoritmo es que la cuenta puede alterarse por un error de transmisión. Por ejemplo, si la cuenta de caracteres de 5 en la segunda trama de la figura 3-4(b) se vuelve un 7, el destino perderá la sincronía y será incapaz de localizar el inicio de la siguiente trama. Incluso si el destino sabe que la trama está mal porque la suma de verificación es incorrecta, no tiene forma de saber dónde comienza la siguiente trama. Regresar una trama a la fuente solicitando una retransmisión tampoco ayuda, ya que el destino no sabe cuántos caracteres tiene que saltar para

llegar al inicio de la retransmisión. Por esta razón, en la actualidad casi no se utiliza el método de conteo de caracteres.

El segundo método de entramado evita el problema de tener que sincronizar nuevamente después de un error, haciendo que cada trama inicie y termine con bytes especiales. En el pasado, los bytes de inicio y final eran diferentes, pero en los años recientes la mayoría de los protocolos han utilizado el mismo byte, llamado **bandera** (o indicador), como delimitador de inicio y final, que en la figura 3-5(a) se muestra como FLAG. De esta manera, si el receptor pierde la sincronía, simplemente puede buscar la bandera para encontrar el final e inicio de la trama actual. Dos banderas consecutivas señalan el final de una trama y el inicio de la siguiente.

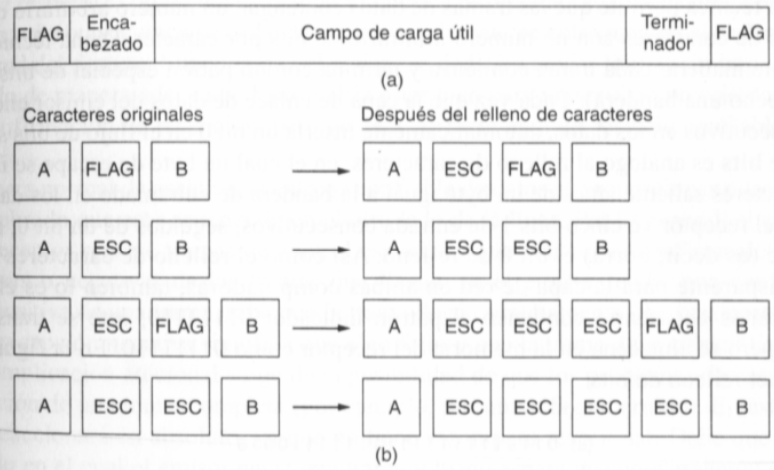


Figura 3-5. (a) Una trama delimitada por banderas. (b) Cuatro ejemplos de secuencias de bytes antes y después del relleno de caracteres.

Cuando se utiliza este método para transmitir datos binarios, como programas objeto o números de punto flotante, surge un problema serio. Se puede dar el caso con mucha facilidad de que el patrón de bits de la bandera aparezca en los datos (*payload*), lo que interferiría en el entramado. Una forma de resolver este problema es hacer que la capa de enlace de datos del emisor inserte un byte de escape especial (ESC) justo antes de cada bandera "accidental" en los datos. La capa de enlace de datos del lado receptor quita el byte de escape antes de entregar los datos a la capa de red. Esta técnica se llama **relleno de caracteres.** Por lo tanto, una bandera de entramado se puede distinguir de uno en los datos por la ausencia o presencia de un byte de escape que la antecede.

Por supuesto que surge la pregunta de qué sucede si un byte de escape aparece en medio de los datos. La respuesta es que también se rellena con un byte de escape. Por lo tanto, cualquier byte de escape individual es parte de una secuencia de escape, mientras que uno doble indica que un

escape sencillo apareció de manera natural en los datos. En la figura 3-5(b) se muestran algunos ejemplos. En todos los casos, la secuencia de bytes que se entrega después de la eliminación de los bytes de escape es exactamente la misma que la secuencia de bytes original.

El esquema de relleno de caracteres que se muestra en la figura 3-5 es una ligera simplificación del esquema empleado en el protocolo PPP que la mayoría de las computadoras utiliza para comunicarse con el proveedor de servicios de Internet. Más tarde analizaremos este protocolo.

Una desventaja importante del uso de esta técnica de entramado es que está fuertemente atada a los caracteres de 8 bits. No todos los códigos utilizan caracteres de 8 bits. Por ejemplo, UNICODE utiliza caracteres de 16 bits. A medida que se desarrollaron las redes, las desventajas de incorporar la longitud del código de caracteres en el mecanismo de entramado se volvieron más obvias, por lo que tuvo que desarrollarse una técnica nueva que permitiera caracteres de tamaño arbitrario.

La nueva técnica permite que las tramas de datos contengan un número arbitrario de bits y admite códigos de caracteres con un número arbitrario de bits por carácter. Dicha técnica funciona de la siguiente manera: cada trama comienza y termina con un patrón especial de bits, 01111110 (que es de hecho una bandera). Cada vez que la capa de enlace de datos del emisor encuentra cinco unos consecutivos en los datos, automáticamente inserta un bit 0 en el flujo de bits saliente. Este **relleno de bits** es análogo al relleno de caracteres, en el cual un byte de escape se inserta en el flujo de caracteres saliente antes de un byte igual a la bandera de entramado en los datos.

Cuando el receptor ve cinco bits 1 de entrada consecutivos, seguidos de un bit 0, automáticamente extrae (es decir, borra) el bit 0 de relleno. Así como el relleno de caracteres es completamente transparente para la capa de red en ambas computadoras, también lo es el relleno de bits. Si los datos de usuario contienen el patrón indicador 01111110, éste se transmite como 011111010, pero se almacena en la memoria del receptor como 01111110. En la figura 3-6 se da un ejemplo del relleno de bits.

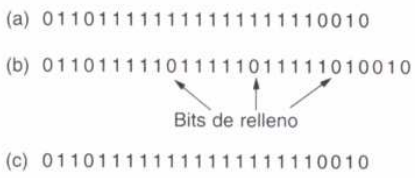


Figura 3-6. Relleno de bits. (a) Los datos originales. (b) Los datos, según aparecen en la línea. (c) Los datos, como se guardan en la memoria del receptor tras eliminar el relleno.

Con el relleno de bits, el límite entre las dos tramas puede ser reconocido sin ambigüedades mediante el patrón de banderas. De esta manera, si el receptor pierde la pista de dónde está, todo lo que tiene que hacer es explorar la entrada en busca de secuencias de banderas, pues sólo pueden ocurrir en los límites de las tramas y nunca en los datos.

El último método de entramado sólo se aplica a las redes en las que la codificación en el medio físico contiene cierta redundancia. Por ejemplo, algunas LANs codifican un bit de datos usando dos bits físicos. Normalmente, un bit 1 es un par alto-bajo, y un bit 0 es un par bajo-alto. El esquema implica que cada bit de datos tiene una transición a medio camino, lo que hace fácil

para el receptor localizar los límites de los bits. Las combinaciones alto-alto y bajo-bajo no se usan para datos, pero en algunos protocolos se utilizan para delimitar tramas.

Como nota final sobre el entramado, muchos protocolos de enlace de datos usan, por seguridad, una combinación de cuenta de caracteres con uno de los otros métodos. Cuando llega una trama, se usa el campo de cuenta para localizar el final de la trama. Sólo si el delimitador apropiado está presente en esa posición y la suma de verificación es correcta, la trama se acepta como válida. De otra manera, se explora el flujo de entrada en busca del siguiente delimitador.

3.1.3 Control de errores

Una vez resuelto el problema de marcar el inicio y el final de cada trama, llegamos al siguiente problema: cómo asegurar que todas las tramas realmente se entreguen en el orden apropiado a la capa de red del destino. Suponga que el emisor se dedicó a enviar tramas sin importarle si estaban llegando de manera adecuada. Esto podría estar bien para un servicio no orientado a la conexión sin confirmación de recepción, pero no será correcto para un servicio confiable orientado a la conexión.

La manera normal de asegurar la entrega confiable de datos es proporcionar retroalimentación al emisor sobre lo que está ocurriendo en el otro lado de la línea. Por lo general, el protocolo exige que el receptor regrese tramas de control especiales que contengan confirmaciones de recepción positivas o negativas de las tramas que llegan. Si el emisor recibe una confirmación de recepción positiva de una trama, sabe que la trama llegó correctamente. Por otra parte, una confirmación de recepción negativa significa que algo falló y que la trama debe transmitirse otra vez.

Una complicación adicional surge de la posibilidad de que los problemas de hardware causen la desaparición de una trama completa (por ejemplo, por una ráfaga de ruido). En este caso, el receptor no reaccionará en absoluto, ya que no tiene razón para reaccionar. Debe quedar claro que un protocolo en el cual el emisor envía una trama y luego espera una confirmación de recepción, positiva o negativa, se quedaría esperando eternamente si se pierde por completo una trama debido a una falla de hardware.

Esta posibilidad se maneja introduciendo temporizadores en la capa de enlace de datos. Cuando el emisor envía una trama, por lo general también inicia un temporizador. Éste se ajusta de modo que expire cuando haya transcurrido un intervalo suficiente para que la trama llegue a su destino, se procese ahí y la confirmación de recepción se regrese al emisor. Por lo general, la trama se recibirá de manera correcta y la confirmación de recepción llegará antes de que el temporizador expire, en cuyo caso se cancelará.

Sin embargo, si la trama o la confirmación de recepción se pierden, el temporizador expirará, alertando al emisor sobre un problema potencial. La solución obvia es simplemente transmitir de nuevo la trama. Sin embargo, aunque las tramas pueden transmitirse muchas veces, existe el peligro de que el receptor acepte la misma trama dos o más veces y que la pase a la capa de red más de una vez. Para evitar que esto ocurra, generalmente es necesario asignar números de secuencia a las tramas que salen, a fin de que el receptor pueda distinguir las retransmisiones de los originales.

El asunto de la administración de temporizadores y números de secuencia para asegurar que cada trama llegue finalmente a la capa de red en el destino una sola vez, ni más ni menos, es una parte importante de las tareas de la capa de enlace de datos. Posteriormente en este capítulo estudiaremos la manera en que se lleva a cabo esta administración, observando una serie de ejemplos de complejidad creciente.

3.1.4 Control de flujo

Otro tema de diseño importante que se presenta en la capa de enlace de datos (y también en las capas superiores) es qué hacer con un emisor que quiere transmitir tramas de manera sistemática y a mayor velocidad que aquella con que puede aceptarlos el receptor. Esta situación puede ocurrir fácilmente cuando el emisor opera en una computadora rápida (o con baja carga) y el receptor opera en una máquina lenta (o sobrecargada). El emisor envía las tramas a alta velocidad hasta que satura por completo al receptor. Aunque la transmisión esté libre de errores, en cierto punto el receptor simplemente no será capaz de manejar las tramas conforme lleguen y comenzará a perder algunas. Es obvio que tiene que hacerse algo para evitar esta situación.

Por lo general se utilizan dos métodos. En el primero, el **control de flujo basado en retroalimentación,** el receptor regresa información al emisor autorizándolo para enviar más datos o indicándole su estado. En el segundo, el **control de flujo basado en tasa,** el protocolo tiene un mecanismo integrado que limita la tasa a la que el emisor puede transmitir los datos, sin recurrir a retroalimentación por parte del receptor. En este capítulo estudiaremos el método de control de flujo basado en retroalimentación debido a que el método basado en tasa no se utiliza en la capa de enlace de datos. En el capítulo 5 analizaremos el método basado en tasa.

Se conocen varios esquemas de control de flujo basados en retroalimentación, pero la mayoría se fundamenta en el mismo principio. El protocolo contiene reglas bien definidas respecto al momento en que un emisor puede enviar la siguiente trama. Con frecuencia estas reglas prohíben el envío de tramas hasta que el receptor lo autorice, implícita o explícitamente. Por ejemplo, cuando se establece una conexión, el receptor podría decir: "Puedes enviarme *n* tramas ahora, pero una vez que lo hagas, no envíes nada más hasta que te indique que continúes". Mas adelante analizaremos los detalles.

3.2 DETECCIÓN Y CORRECCIÓN DE ERRORES

Como vimos en el capítulo 2, el sistema telefónico tiene tres partes: los conmutadores, las troncales interoficinas y los circuitos locales. Las primeras dos son ahora casi enteramente digitales en la mayoría de los países desarrollados. Los circuitos locales aún son cables de par trenzado de cobre analógicos en todos lados y continuarán así durante décadas debido al enorme costo de su reemplazo. Aunque los errores son raros en la parte digital, aún son comunes en los circuitos locales. Además, la comunicación inalámbrica se está volviendo más común, y las tasas de errores son de magnitud mucho mayor que en las troncales de fibra interoficinas. La conclusión es: los errores de transmisión van a ser inevitables durante muchos años más. Tendremos que aprender a lidiar con ellos.

Como resultado de los procesos físicos que los generan, los errores en algunos medios (por ejemplo, la radio) tienden a aparecer en ráfagas y no de manera individual. El hecho de que los errores lleguen en ráfaga tiene ventajas y desventajas con respecto a los errores aislados de un solo bit. Por el lado de las ventajas, los datos de computadora siempre se envían en bloques de bits. Suponga que el tamaño de bloque es de 1000 bits y la tasa de errores es de 0.001 por bit. Si los errores fueran independientes, la mayoría de los bloques contendría un error. Sin embargo, si los errores llegan en ráfagas de 100, en promedio sólo uno o dos bloques de cada 100 serán afectados. La desventaja de los errores en ráfaga es que son mucho más difíciles de detectar y corregir que los errores aislados.

3.2.1 Códigos de corrección de errores

Los diseñadores de redes han desarrollado dos estrategias principales para manejar los errores. Una es incluir suficiente información redundante en cada bloque de datos transmitido para que el receptor pueda deducir lo que debió ser el carácter transmitido. La otra estrategia es incluir sólo suficiente redundancia para permitir que el receptor sepa que ha ocurrido un error (pero no qué error) y entonces solicite una retransmisión. La primera estrategia utiliza **códigos de corrección de errores;** la segunda usa **códigos de detección de errores.** El uso de códigos de corrección de errores usualmente se conoce como **corrección de errores hacia adelante.**

Cada una de estas técnicas ocupa un nicho ecológico diferente. En los canales que son altamente confiables, como los de fibra, es más económico utilizar un código de detección de errores y simplemente retransmitir los bloques defectuosos que surgen ocasionalmente. Sin embargo, en los canales que causan muchos errores, como los enlaces inalámbricos, es mejor agregar la redundancia suficiente a cada bloque para que el receptor pueda descubrir cuál era el bloque original transmitido, en lugar de confiar en una retransmisión, que también podría tener errores.

Para entender la manera en que pueden manejarse los errores, es necesario estudiar de cerca lo que es en realidad un error. Por lo general, una trama consiste en m bits de datos (es decir, de mensaje) y r bits redundantes o de verificación. Sea la longitud total n (es decir, $n = m + r$). A una unidad de n bits que contiene datos y bits de verificación se le conoce como **palabra codificada** de n bits.

Dadas dos palabras codificadas cualesquiera, digamos 10001001 y 10110001, es posible determinar cuántos bits correspondientes difieren. En este caso, difieren tres bits. Para determinar la cantidad de bits diferentes, basta aplicar un OR exclusivo a las dos palabras codificadas y contar la cantidad de bits 1 en el resultado, por ejemplo:

```
10001001
10110001
00111000
```

La cantidad de posiciones de bits en la que difieren dos palabras codificadas se llama **distancia de Hamming** (Hamming, 1950). Su significado es que, si dos palabras codificadas están separadas una distancia de Hamming d, se requerirán d errores de un bit para convertir una en la otra.

En la mayoría de las aplicaciones de transmisión de datos, todos los 2^m mensajes de datos posibles son legales, pero debido a la manera en que se calculan los bits de verificación no se usan todas las 2^n palabras codificadas posibles. Dado el algoritmo de cálculo de los bits de verificación, es posible construir una lista completa de palabras codificadas legales y encontrar, en esta lista, las dos palabras codificadas cuya distancia de Hamming es mínima. Ésta es la distancia de Hamming de todo el código.

Las propiedades de detección y corrección de errores de un código dependen de su distancia de Hamming. Para detectar d errores se necesita un código con distancia $d + 1$, pues con tal código no hay manera de que d errores de un bit puedan cambiar una palabra codificada válida a otra. Cuando el receptor ve una palabra codificada no válida, sabe que ha ocurrido un error de transmisión. De manera similar, para corregir d errores se necesita un código de distancia $2d + 1$, pues así las palabras codificadas legales están tan separadas que, aun con d cambios, la palabra codificada original sigue estando más cercana que cualquier otra palabra codificada, por lo que puede determinarse de manera única.

Como ejemplo sencillo de código de detección de errores, considere un código en el que se agrega un solo **bit de paridad** a los datos. Este bit se escoge de manera que la cantidad de bits 1 en la palabra código sea par (o impar). Por ejemplo, cuando se envía 1011010 con paridad par, se agrega un bit al final, y se vuelve 10110100. Con paridad impar, 1011010 se vuelve 10110101. Un código con un solo bit de paridad tiene una distancia de 2, pues cualquier error de un bit produce una palabra codificada con la paridad equivocada. Este sistema puede usarse para detectar errores individuales.

Como ejemplo sencillo de código de corrección de errores, considere un código con sólo cuatro palabras codificadas válidas:

0000000000, 0000011111, 1111100000 y 1111111111

Este código tiene una distancia de 5, lo que significa que puede corregir errores dobles. Si llega la palabra codificada 0000000111, el receptor sabe que el original debió ser 0000011111. Sin embargo, si un error triple cambia 0000000000 a 0000000111, el error no se corregirá de manera adecuada.

Imagine que deseamos diseñar un código con m bits de mensaje y r bits de verificación que permitirá la corrección de todos los errores individuales. Cada uno de los 2^m mensajes legales tiene n palabras codificadas ilegales a una distancia 1 de él. Éstas se forman invirtiendo en forma sistemática cada uno de los n bits de la palabra codificada de n bits que la forman. Por lo tanto, cada uno de los 2^m mensajes legales requiere $n + 1$ patrones de bits dedicados a él. Dado que la cantidad de patrones de bits es 2^n, debemos tener $(n + 1)2^m \leq 2^n$. Usando $n = m + r$, este requisito se vuelve $(m + r + 1) \leq 2^r$. Dado m, esto impone un límite inferior a la cantidad de bits de verificación necesarios para corregir errores individuales.

De hecho, este límite inferior teórico puede lograrse usando un método gracias a Hamming (1950). Los bits de la palabra codificada se numeran en forma consecutiva, comenzando por el bit 1 a la izquierda, el bit 2 a su derecha inmediata, etcétera. Los bits que son potencias de 2 (1, 2, 4, 8, 16, etcétera) son bits de verificación. El resto (3, 5, 6, 7, 9, etcétera) se rellenan con los m bits de datos. Cada bit de verificación obliga a que la paridad de un grupo de bits, incluyéndolo a él

mismo, sea par (o impar). Un bit puede estar incluido en varios cálculos de paridad. Para ver a qué bits de verificación contribuye el bit de datos en la posición k, reescriba k como una suma de potencias de 2. Por ejemplo, $11 = 1 + 2 + 8$ y $29 = 1 + 4 + 8 + 16$. Se comprueba un bit solamente por los bits de verificación que ocurren en su expansión (por ejemplo, el bit 11 es comprobado por los bits 1, 2 y 8).

Cuando llega una palabra codificada, el receptor inicializa a cero un contador y luego examina cada bit de verificación, k ($k = 1, 2, 4, 8,...$), para ver si tiene la paridad correcta. Si no, suma k al contador. Si el contador es igual a cero tras haber examinado todos los bits de verificación (es decir, si todos fueron correctos), la palabra codificada se acepta como válida. Si el contador es diferente de cero, contiene el número del bit incorrecto. Por ejemplo, si los bits de verificación 1, 2 y 8 tienen errores, el bit invertido es el 11, pues es el único comprobado por los bits 1, 2 y 8. En la figura 3-7 se muestran algunos caracteres ASCII de 7 bits codificados como palabras codificadas de 11 bits usando un código de Hamming. Recuerde que los datos se encuentran en las posiciones de bit 3, 5, 6, 7, 9, 10 y 11.

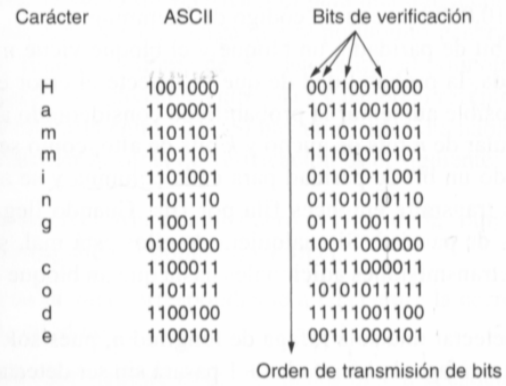


Figura 3-7. Uso de un código de Hamming para corregir errores en ráfaga.

Los códigos de Hamming sólo pueden corregir errores individuales. Sin embargo, hay un truco que puede servir para que los códigos de Hamming corrijan errores de ráfaga. Se dispone como matriz una secuencia de k palabras codificadas consecutivas, con una palabra codificada por fila. Normalmente se transmitiría una palabra codificada a la vez, de izquierda a derecha. Para corregir los errores en ráfaga, los datos deben transmitirse una columna a la vez, comenzando por la columna del extremo izquierdo. Cuando todos los bits k han sido enviados, se envía la segunda columna, y así sucesivamente. Cuando la trama llega al receptor, la matriz se reconstruye, una columna a la vez. Si ocurre un error en ráfaga de longitud k, cuando mucho se habrá afectado 1 bit de cada una de las k palabras codificadas; sin embargo, el código de Hamming puede corregir un error por palabra codificada, así que puede restaurarse la totalidad del bloque. Este método usa kr bits de verificación para inmunizar bloques de km bits de datos contra un solo error en ráfaga de longitud k o menos.

3.2.2 Códigos de detección de errores

Los códigos de corrección de errores se utilizan de manera amplia en los enlaces inalámbricos, que son notoriamente más ruidosos y propensos a errores que el alambre de cobre o la fibra óptica. Sin los códigos de corrección de errores sería difícil pasar cualquier cosa. Sin embargo, a través del cable de cobre o de la fibra óptica, la tasa de error es mucho más baja, por lo que la detección de errores y la retransmisión por lo general son más eficientes ahí para manejar un error ocasional.

Como un ejemplo simple, considere un canal en el que los errores son aislados y la tasa de errores es de 10^{-6} por bit. Sea el tamaño de bloque 1000 bits. Para proporcionar corrección de errores en bloques de 1000 bits se requieren 10 bits de verificación; un megabit de datos requerirá 10000 bits de verificación. Para detectar un solo bloque con 1 bit de error, basta con un bit de paridad por bloque. Por cada 1000 bloques se tendrá que transmitir un bloque extra (1001 bits). La sobrecarga total del método de detección de errores + retransmisión es de sólo 2001 bits por megabit de datos, contra 10,000 bits con un código de Hamming.

Si se agrega un solo bit de paridad a un bloque y el bloque viene muy alterado por una ráfaga de errores prolongada, la probabilidad de que se detecte el error es de 0.5, lo que difícilmente es aceptable. Es posible aumentar la probabilidad considerando a cada bloque por enviar como una matriz rectangular de n bits de ancho y k bits de alto, como se describió anteriormente. Se calcula por separado un bit de paridad para cada columna y se agrega a la matriz como última fila. La matriz se transmite entonces fila por fila. Cuando llega el bloque, el receptor comprueba todos los bits de paridad. Si cualquiera de ellos está mal, solicita la retransmisión del bloque. Se solicitan retransmisiones adicionales hasta que un bloque entero se reciba sin ningún error de paridad.

Este método puede detectar una sola ráfaga de longitud n, pues sólo se cambiará un bit por columna. Sin embargo, una ráfaga de longitud $n + 1$ pasará sin ser detectada si se invierten el primero y último bits, y si todos los demás bits están correctos. (Una ráfaga de errores no implica que todos los bits están mal; sólo implica que cuando menos el primero y el último están mal.) Si el bloque está muy alterado por una ráfaga continua o por múltiples ráfagas más cortas, la probabilidad de que cualquiera de las n columnas tenga, por accidente, la paridad correcta es de 0.5, por lo que la probabilidad de aceptar un bloque alterado cuando no se debe es de 2^{-n}.

Aunque en algunos casos el método anterior puede ser adecuado, en la práctica se usa uno muy definido: el **código polinomial** (también conocido como **código de redundancia cíclica** o **código CRC**). Los códigos polinomiales se basan en el tratamiento de cadenas de bits como representaciones de polinomios con coeficientes de 0 y 1 solamente. Una trama de k bits se considera como la lista de coeficientes de un polinomio con k términos que van de x^{k-1} a x^0. Se dice que tal polinomio es de grado $k - 1$. El bit de orden mayor (que se encuentra más a la izquierda) es el coeficiente de x^{k-1}, el siguiente bit es el coeficiente de x^{k-2} y así sucesivamente. Por ejemplo, 110001 tiene 6 bits y, por lo tanto, representa un polinomio de seis términos con coeficientes 1, 1, 0, 0, 0 y 1: $x^5 + x^4 + x^0$.

La aritmética polinomial se hace mediante una operación módulo 2, de acuerdo con las reglas de la teoría de campos algebraicos. No hay acarreos para la suma, ni préstamos para la resta. Tanto la suma como la resta son idénticas a un OR exclusivo. Por ejemplo:

10011011	00110011	11110000	01010101
+11001010	+11001101	−10100110	−10101111
01010001	11111110	01010110	11111010

La división se lleva a cabo de la misma manera que en binario, excepto que la resta es módulo 2, igual que antes. Se dice que un divisor "cabe" en un dividendo si éste tiene tantos bits como el divisor.

Cuando se emplea el método de código polinomial, el emisor y el receptor deben acordar por adelantado un **polinomio generador,** $G(x)$. Tanto los bits de orden mayor y menor del generador deben ser 1. Para calcular la **suma de verificación** para una trama con m bits, correspondiente al polinomio $M(x)$, la trama debe ser más larga que el polinomio generador. La idea es incluir una suma de verificación al final de la trama de tal manera que el polinomio representado por la trama con suma de verificación sea divisible entre $G(x)$. Cuando el receptor recibe la trama con suma de verificación, intenta dividirla entre $G(x)$. Si hay un residuo, ha habido un error de transmisión.

El algoritmo para calcular la suma de verificación es el siguiente:

1. Sea r el grado de $G(x)$. Anexe r bits cero al final de la trama, para que ahora contenga $m + r$ bits y corresponda al polinomio $x^r M(x)$.
2. Divida la cadena de bits correspondiente a $G(x)$ entre la correspondiente a $x^r M(x)$ usando una división módulo 2.
3. Reste el residuo (que siempre es de r o menos bits) a la cadena de bits correspondiente a $x^r M(x)$ usando una resta módulo 2. El resultado es la trama con suma de verificación que va a transmitirse. Llame a su polinomio $T(x)$.

En la figura 3-8 se ilustra el cálculo para una trama 1101011011 utilizando el generador $G(x) = x^4 + x + 1$.

Debe quedar claro que $T(x)$ es divisible (módulo 2) entre $G(x)$. En cualquier problema de división, si se resta el residuo del dividendo, lo que queda es divisible entre el divisor. Por ejemplo, en base 10, si se divide 210,278 entre 10,941, el residuo es 2399. Si se resta 2399 a 210,278, lo que queda (207,879) es divisible entre 10,941.

Ahora analizaremos el alcance de este método. ¿Qué tipos de error se detectarán? Imagine que ocurre un error de transmisión tal que en lugar de que llegue la cadena de bits para $T(x)$, llega $T(x) + E(x)$. Cada bit 1 en $E(x)$ corresponde a un bit que ha sido invertido. Si hay k bits 1 en $E(x)$, han ocurrido k errores de un solo bit. Una ráfaga de errores individual se caracteriza por un 1 inicial, una mezcla de ceros y unos, y un 1 final, siendo los demás bits 0.

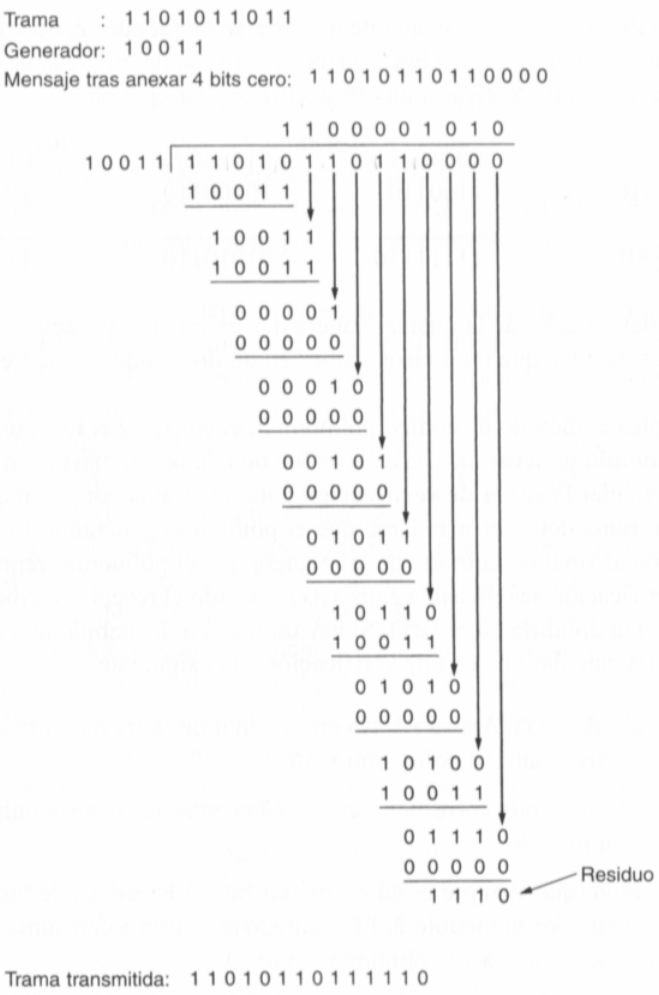


Figura 3-8. Cálculo de la suma de verificación de código polinomial.

Al recibir la trama con suma de verificación, el receptor la divide entre $G(x)$; es decir, calcula $[T(x) + E(x)]/G(x)$. $T(x)/G(x)$ es 0, por lo que el resultado del cálculo es simplemente $E(x)/G(x)$. No se detectarán los errores que por casualidad correspondan a polinomios que contengan $G(x)$ como factor; todos los demás errores serán atrapados.

Si ha ocurrido un error de un solo bit, $E(x) = x^i$, donde i determina qué bit es erróneo. Si $G(x)$ contiene dos o más términos, nunca será divisor exacto de $E(x)$, por lo que se detectarán los errores de un solo bit.

Si han ocurrido dos errores de un solo bit aislados, $E(x) = x^i + x^j$, donde $i > j$. Esto también se puede escribir como $E(x) = x^j(x^{i-j} + 1)$. Si suponemos que $G(x)$ no es divisible entre x, una condición suficiente para detectar todos los errores dobles es que $G(x)$ no divida a $x^k + 1$ para ninguna k hasta el valor máximo de $i - j$ (es decir, hasta la longitud máxima de la trama). Se conocen polinomios sencillos de bajo grado que dan protección a tramas largas. Por ejemplo, $x^{15} + x^{14} + 1$ no será divisor exacto de $x^k + 1$ para ningún valor de k menor que 32,768.

Si hay una cantidad impar de bits con error, $E(x)$ contiene un número impar de términos (por ejemplo, $x^5 + x^2 + 1$, pero no $x^2 + 1$). Curiosamente, ningún polinomio con un número impar de términos posee a $x + 1$ como un factor en el sistema de módulo 2. Haciendo $x + 1$ un factor de $G(x)$, podemos atrapar todos los errores consistentes en un número impar de bits invertidos.

Para comprobar que ningún polinomio con una cantidad impar de términos es divisible entre $x + 1$, suponga que $E(x)$ tiene un número impar de términos y que es divisible entre $x + 1$. Factorice $E(x)$ en $(x + 1)\ Q(x)$. Ahora evalúe $E(1) = (1 + 1)Q(1)$. Dado que $1 + 1 = 0$ (módulo 2), $E(1)$ debe ser cero. Si $E(x)$ tiene un número impar de términos, la sustitución de 1 por x en cualquier lugar siempre dará como resultado un 1. Por lo tanto, ningún polinomio con un número impar de términos es divisible entre $x + 1$.

Por último, y lo que es más importante, un código polinomial con r bits de verificación detectará todos los errores en ráfaga de longitud $\leq r$. Un error en ráfaga de longitud k puede representarse mediante $x^i(x^{k-1} + \ldots + 1)$, donde i determina la distancia a la que se encuentra la ráfaga desde el extremo derecho de la trama recibida. Si $G(x)$ contiene un término x^0, no tendrá x^i como factor, por lo que, si el grado de la expresión entre paréntesis es menor que el grado de $G(x)$, el residuo nunca puede ser cero.

Si la longitud de la ráfaga es de $r + 1$, el residuo de la división entre $G(x)$ será cero si, y sólo si, la ráfaga es idéntica a $G(x)$. Por la definición de ráfaga, el primero y el último bit deben ser 1, así que el que sean iguales o no depende de los $r - 1$ bits intermedios. Si se consideran igualmente probables todas las combinaciones, la probabilidad de que se acepte como válida tal trama incorrecta es de $1/2^{r-1}$.

También puede demostrarse que cuando ocurre una ráfaga de errores mayor que $r + 1$, o cuando ocurren varias ráfagas más cortas, la probabilidad de que una trama incorrecta no sea detectada es de $1/2^r$, suponiendo que todos los patrones de bits sean igualmente probables.

Ciertos polinomios se han vuelto estándares internacionales. El que se utiliza en el IEEE 802 es:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

Entre otras propiedades deseables, tiene la de que detecta todas las ráfagas con una longitud de 32 o menor y todas las ráfagas que afecten un número impar de bits.

Aunque el cálculo requerido para obtener la suma de verificación puede parecer complicado, Peterson y Brown (1961) han demostrado que puede construirse un circuito sencillo con un registro de desplazamiento para calcular y comprobar las sumas de verificación por hardware. En la práctica, casi siempre se usa este hardware. La mayoría de las LANs lo utiliza y, en algunos casos, también lo hacen las líneas punto a punto.

Durante décadas se ha supuesto que las tramas para las que se generan sumas de verificación contienen bits aleatorios. Todos los análisis de algoritmos de suma de verificación se han hecho bajo este supuesto. En fechas más recientes, la inspección de datos reales ha demostrado que este supuesto es equivocado. Como consecuencia, en algunas circunstancias los errores no detectados son mucho más comunes de lo que se pensaba anteriormente (Partridge y cols., 1995).

3.3 PROTOCOLOS ELEMENTALES DE ENLACE DE DATOS

Como introducción al tema de los protocolos, comenzaremos por estudiar tres protocolos de complejidad creciente. Los lectores interesados pueden conseguir un simulador de estos protocolos y otros subsecuentes a través de WWW (vea el prefacio). Antes de estudiar los protocolos, es útil hacer explícitos algunos de los supuestos implícitos del modelo de comunicaciones. Para comenzar, estamos suponiendo que en las capas física, de enlace de datos y de red hay procesos independientes que se comunican pasando mensajes de un lado a otro. En muchos casos, los procesos de las capas física y de enlace de datos se ejecutan en un procesador dentro de un chip especial de E/S y los de la capa de red lo hacen en la CPU principal. Sin embargo, también puede haber otras implementaciones (por ejemplo, tres procesos en un solo chip de E/S o las capas física y de enlace de datos como procedimientos invocados por el proceso de la capa de red). En cualquier caso, el hecho de tratar las tres capas como procesos independientes hace más nítido el análisis en el terreno conceptual y también sirve para subrayar la independencia de las capas.

Otro supuesto clave es que la máquina *A* desea mandar un flujo considerable de datos a la máquina *B* usando un servicio confiable orientado a la conexión. Después consideraremos el caso en que *B* también quiere mandar datos a *A* de manera simultánea. Se ha supuesto que *A* tiene un suministro infinito de datos listos para ser enviados y nunca tiene que esperar a que se produzcan datos. Cuando la capa de enlace de datos de *A* solicita datos, la capa de red siempre es capaz de proporcionarlos de inmediato. (Esta restricción también se desechará posteriormente.)

También supondremos que las máquinas no fallan. Es decir, estos protocolos manejan errores de comunicación, pero no los problemas causados por computadoras que fallan y se reinician.

En lo que concierne a la capa de enlace de datos, el paquete que se le pasa a través de la interfaz desde la capa de red es de datos puros, que deben ser entregados bit por bit a la capa de red del destino. El hecho de que la capa de red del destino pueda interpretar parte del paquete como un encabezado no es de importancia para la capa de enlace de datos.

Cuando la capa de enlace de datos acepta un paquete, lo encapsula en una trama agregándole un encabezado y un terminador de enlace de datos (vea la figura 3-1). Por lo tanto, una trama consiste en un paquete incorporado, cierta información de control (en el encabezado) y una suma de verificación (en el terminador). A continuación la trama se transmite a la capa de enlace de datos de la otra máquina. Supondremos que existen procedimientos de biblioteca adecuados *to_physical_layer* para enviar una trama y *from_physical_layer* para recibir una trama. El hardware emisor calcula y agrega la suma de verificación (y de esta manera crea el terminador) por lo que el software

de la capa de enlace de datos no necesita preocuparse por ella. Por ejemplo, podría utilizarse el algoritmo polinomial analizado antes en este capítulo.

Inicialmente el receptor no tiene nada que hacer. Sólo está esperando que ocurra algo. En los protocolos de ejemplo de este capítulo indicamos que la capa de enlace de datos está en espera de que ocurra algo con la llamada de procedimiento *wait_for_event*(*&event*). Este procedimiento sólo regresa cuando ocurre algo (por ejemplo, cuando llega una trama). Al regresar, la variable *event* indica lo que ha ocurrido. El grupo de eventos posibles difiere para cada uno de los diferentes protocolos que describiremos, y se definirán por separado para cada protocolo. Observe que en una situación más realista, la capa de enlace de datos no se quedará en un ciclo cerrado esperando un evento, como hemos sugerido, sino que recibirá una interrupción, la que ocasionará que suspenda lo que estaba haciendo y proceda a manejar la trama entrante. Sin embargo, por sencillez ignoraremos todos los detalles de la actividad paralela en la capa de enlace de datos y daremos por hecho que la capa está dedicada de tiempo completo a manejar nuestro canal.

Cuando llega una trama al receptor, el hardware calcula la suma de verificación. Si ésta es incorrecta (es decir, si hubo un error de transmisión), se le informa a la capa de enlace de datos (*event* = *cksum_err*). Si la trama entrante llega sin daño, también se le informa a la capa de enlace de datos (*event* = *frame_arrival*) para que pueda adquirir la trama para inspeccionarla usando *from_physical_layer*. Tan pronto como la capa de enlace de datos receptora adquiere una trama sin daños, revisa la información de control del encabezado y, si todo está bien, pasa la parte que corresponde al paquete a la capa de red. En ninguna circunstancia se entrega un encabezado de trama a la capa de red.

Hay una buena razón por la que la capa de red nunca debe recibir ninguna parte del encabezado de trama: para mantener completamente separados el protocolo de red y el de enlace de datos. En tanto la capa de red no sepa nada en absoluto sobre el protocolo de enlace de datos ni el formato de la trama, éstos podrán cambiarse sin requerir cambios en el software de la capa de red. Al proporcionarse una interfaz rígida entre la capa de red y la de enlace de datos se simplifica en gran medida el diseño del software, pues los protocolos de comunicación de las diferentes capas pueden evolucionar en forma independiente.

En la figura 3-9 se muestran algunas declaraciones comunes (en C) para muchos de los protocolos que se analizarán después. Allí se definen cinco estructuras de datos: *boolean*, *seq_nr*, *packet*, *frame_kind* y *frame*. Un *boolean* es un tipo de dato numérico que puede tener los valores *true* y *false*. Un *seq_nr* (número de secuencia) es un entero pequeño que sirve para numerar las tramas, a fin de distinguirlas. Estos números de secuencia van de 0 hasta *MAX_SEQ* (inclusive), que se define en cada protocolo que lo necesita. Un *packet* es la unidad de intercambio de información entre la capa de red y la de enlace de datos en la misma máquina, o entre entidades iguales de la capa de red. En nuestro modelo siempre contiene *MAX_PKT* bytes, pero en la práctica sería de longitud variable.

Un *frame* está compuesto de cuatro campos: *kind*, *Seq*, *ack* e *info*. Los primeros tres contienen información de control y el último puede contener los datos por transferir. Estos campos de control constituyen en conjunto el **encabezado de la trama.**

```
#define MAX_PKT 1024                                /* determina el tamaño del paquete
                                                       en bytes */

typedef enum {false, true} boolean;                 /* tipo booleano */
typedef unsigned int seq_nr;                        /* números de secuencia o
                                                       confirmación */
typedef struct {unsigned char data[MAX_PKT];} packet;/* definición de paquete */
typedef enum {data, ack, nak} frame_kind;           /* definición de frame_kind */
typedef struct {                                    /* las tramas se transportan en
                                                       esta capa */
 frame_kind kind;                                   /* ¿qué clase de trama es? */
 seq_nr seq;                                        /* número de secuencia */
 seq_nr ack;                                        /* número de confirmación de
                                                       recepción */
 packet info;                                       /* paquete de la capa de red */
} frame;

/* Espera que ocurra un evento; devuelve el tipo en la variable event. */
void wait_for_event(event_type *event);

/* Obtiene un paquete de la capa de red para transmitirlo por el canal. */
void from_network_layer(packet *p);

/* Entrega información de una trama entrante a la capa de red. */
void to_network_layer(packet *p);

/* Obtiene una trama entrante de la capa física y la copia en r. */
void from_physical_layer(frame *r);

/* Pasa la trama a la capa física para transmitirla. */
void to_physical_layer(frame *s);

/* Arranca el reloj y habilita el evento de expiración de temporizador. */
void start_timer(seq_nr k);

/* Detiene el reloj e inhabilita el evento de expiración de temporizador. */
void stop_timer(seq_nr k);

/* Inicia un temporizador auxiliar y habilita el evento ack_timeout. */
void start_ack_timer(void);

/* Detiene el temporizador auxiliar e inhabilita el evento ack_timeout. */
void stop_ack_timer(void);

/* Permite que la capa de red cause un evento network_layer_ready. */
void enable_network_layer(void);

/* Evita que la capa de red cause un evento network_layer_ready. */
void disable_network_layer(void);

/* La macro inc se expande en línea: incrementa circularmente k. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

Figura 3-9. Algunas definiciones necesarias en los protocolos que siguen. Estas definiciones se encuentran en el archivo *protocol.h*.

El campo *kind* indica si hay datos en la trama, porque algunos de los protocolos distinguen entre las tramas que contienen exclusivamente información de control y los que también contienen datos. Los campos *seq* y *ack* se emplean para números de secuencia y confirmaciones de recepción, respectivamente; su uso se describirá posteriormente con mayor detalle. El campo *info* de una trama de datos contiene un solo paquete; el campo *info* de una trama de control no se usa. En una implementación más realista se usaría un campo *info* de longitud variable, omitiéndolo por completo en las tramas de control.

Es importante entender la relación entre un paquete y una trama. La capa de red construye un paquete tomando un mensaje de la capa de transporte y agregándole el encabezado de la capa de red. Este paquete se pasa a la capa de enlace de datos para incluirlo en el campo *info* de una trama saliente. Cuando ésta llega a su destino, la capa de enlace de datos extrae de ella el paquete y a continuación lo pasa a la capa de red. De esta manera, esta capa puede actuar como si las máquinas pudieran intercambiar paquetes directamente.

En la figura 3-9 también se listan varios procedimientos que son rutinas de biblioteca cuyos detalles dependen de la implementación, por lo que no nos ocuparemos de su funcionamiento interno aquí. El procedimiento *wait_for_event* se queda en un ciclo cerrado esperando que algo ocurra, como se mencionó antes. Con los procedimientos *to_network_layer* y *from_network_layer,* la capa de enlace de datos pasa paquetes a la capa de red y acepta paquetes de ella, respectivamente. Observe que *from_physical_layer* y *to_physical_layer* pasan tramas entre la capa de enlace de datos y la capa física, y que los procedimientos *to_network_layer* y *from_network_layer* pasan paquetes entre la capa de enlace de datos y la capa de red. En otras palabras, *to_network_layer* y *from_network_layer* tienen que ver con la interfaz entre las capas 2 y 3, y *from_physical_layer* y *to_physical_layer,* con la interfaz entre las capas 1 y 2.

En la mayoría de los protocolos suponemos un canal inestable que pierde tramas completas ocasionalmente. Para poder recuperarse de tales calamidades, la capa de enlace de datos emisora debe arrancar un temporizador o reloj interno cada vez que envía una trama. Si no obtiene respuesta tras transcurrir cierto intervalo de tiempo predeterminado, el temporizador expira y la capa de enlace de datos recibe una señal de interrupción.

En nuestros protocolos, esto se maneja permitiendo que el procedimiento *wait_for_event* devuelva *event = timeout*. Los procedimientos *start_timer* y *stop_timer* inician y detienen, respectivamente, el temporizador. Las terminaciones del temporizador sólo son posibles cuando éste se encuentra en funcionamiento. Se permite explícitamente llamar a *start_timer* cuando el temporizador está funcionando; tal llamada tan sólo restablece el reloj para hacer que el temporizador termine después de haber transcurrido un intervalo completo de temporización (a menos que se restablezca o apague antes).

Los procedimientos *star_ack_timer* y *stop_ack_timer* controlan un temporizador auxiliar usado para generar confirmaciones de recepción en ciertas condiciones.

Los procedimientos *enable_network_layer* y *disable_network_layer* se usan en los protocolos más complicados, en los que ya no suponemos que la capa de red siempre tiene paquetes que enviar. Cuando la capa de enlace de datos habilita a la capa de red, ésta tiene permitido interrumpir cuando tenga que enviar un paquete. Esto lo indicamos con *event = network_layer_ready*. Cuando una capa de red está inhabilitada, no puede causar tales eventos. Teniendo cuidado respecto a

cuándo habilitar e inhabilitar su capa de red, la capa de enlace de datos puede evitar que la capa de red la sature con paquetes para los que no tiene espacio de búfer.

Los números de secuencia de las tramas siempre están en el intervalo de 0 a *MAX_SEQ* (inclusive), donde *MAX_SEQ* es diferente para los distintos protocolos. Con frecuencia es necesario avanzar circularmente en 1 un número de secuencia (por ejemplo, *MAX_SEQ* va seguido de 0). La macro *inc* lleva a cabo este incremento. Esta función se ha definido como macro porque se usa en línea dentro de la ruta crítica. Como veremos después en este libro, el factor que limita el desempeño de una red con frecuencia es el procesamiento del protocolo, por lo que definir como macros las operaciones sencillas como ésta no afecta la legibilidad del código y sí mejora el desempeño. Además, ya que *MAX_SEQ* tendrá diferentes valores en diferentes protocolos, al hacer que *inc* sea una macro, cabe la posibilidad de incluir todos los protocolos en el mismo binario sin conflictos. Esta capacidad es útil para el simulador.

Las declaraciones de la figura 3-9 son parte de todos los protocolos que siguen. Para ahorrar espacio y proporcionar una referencia práctica, se han extraído y listado juntas, pero conceptualmente deberían estar integradas con los protocolos mismos. En C, esta integración se efectúa poniendo las definiciones en un archivo especial de encabezado, en este caso *protocol.h*, y usando la directiva #include del preprocesador de C para incluirlas en los archivos de protocolos.

3.3.1 Un protocolo símplex sin restricciones

Como ejemplo inicial consideraremos un protocolo que es lo más sencillo posible. Los datos se transmiten sólo en una dirección; las capas de red tanto del emisor como del receptor siempre están listas; el tiempo de procesamiento puede ignorarse; hay un espacio infinito de búfer y, lo mejor de todo, el canal de comunicación entre las capas de enlace de datos nunca tiene problemas ni pierde tramas. Este protocolo completamente irreal, al que apodaremos "utopía", se muestra en la figura 3-10.

El protocolo consiste en dos procedimientos diferentes, uno emisor y uno receptor. El emisor se ejecuta en la capa de enlace de datos de la máquina de origen y el receptor se ejecuta en la capa de enlace de datos de la máquina de destino. No se usan números de secuencia ni confirmaciones de recepción, por lo que no se necesita *MAX_SEQ*. El único tipo de evento posible es *frame_arrival* (es decir, la llegada de una trama sin daños).

El emisor está en un ciclo `while` infinito que sólo envía datos a la línea tan rápidamente como puede. El cuerpo del ciclo consiste en tres acciones: obtener un paquete de la (siempre dispuesta) capa de red, construir una trama de salida usando la variable *s* y enviar la trama a su destino. Este protocolo sólo utiliza el campo *info* de la trama, pues los demás campos tienen que ver con el control de errores y de flujo, y aquí no hay restricciones de control de errores ni de flujo.

El receptor también es sencillo. Inicialmente, espera que algo ocurra, siendo la única posibilidad la llegada de una trama sin daños. En algún momento, la trama llega y el procedimiento *wait_for_event* regresa, conteniendo *event* el valor *frame_arrival* (que de todos modos se ignora). La llamada a *from_physical_layer* elimina la trama recién llegada del búfer de hardware y la

```
/* El protocolo 1 (utopía) provee la transmisión de datos en una sola
dirección, del emisor al  receptor. Se supone que el canal de comunicación
está libre de errores, y que el receptor es capaz de procesar toda la entrada
a una rapidez infinita. En consecuencia, el emisor se  mantiene en un ciclo,
enviando datos a la línea tan rápidamente como puede. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender1(void)
{
  frame s;                          /* búfer para una trama de salida */
  packet buffer;                    /* búfer para un paquete de salida */

  while (true) {
     from_network_layer(&buffer); /* consigue algo que enviar */
     s.info = buffer;             /* lo copia en s para transmisión */
     to_physical_layer(&s);       /* lo envía a su destino */
  }                               /* Mañana, y mañana, y mañana,
                                     Se arrastra a este mísero paso de día
                                       a día
                                     Hasta la última sílaba del tiempo
                                       recordado
                                         -Macbeth, V, v */
}

void receiver1(void)
{
  frame r;
  event_type event;               /* ocupado por wait, pero no se usa aquí */

  while (true) {
     wait_for_event(&event);      /* la única posibilidad es frame_arrival */
     from_physical_layer(&r);     /* obtiene la trama entrante */
     to_network_layer(&r.info);   /* pasa los datos a la capa de red */
  }
}
```

Figura 3-10. Protocolo símplex sin restricciones.

coloca en la variable *r*, en donde el código receptor pueda obtenerla. Por último, la parte de datos se pasa a la capa de red y la capa de enlace de datos se retira para esperar la siguiente trama, suspendiéndose efectivamente hasta que llega la trama.

3.3.2 Protocolo símplex de parada y espera

Ahora omitiremos el supuesto más irreal hecho en el protocolo 1: la capacidad de la capa de red receptora de procesar datos de entrada con una rapidez infinita (o, lo que es equivalente, la presencia en la capa de enlace de datos receptora de una cantidad infinita de espacio de búfer en el cual almacenar todas las tramas de entrada mientras esperan su respectivo turno). Todavía se supone que el canal de comunicaciones está libre de errores y que el tráfico de datos es símplex.

El problema principal que debemos resolver aquí es cómo evitar que el emisor sature al receptor enviando datos a mayor velocidad de la que este último puede procesarlos. En esencia, si el receptor requiere un tiempo Δt para ejecutar *from_physical_layer* más *to_network_layer*, el emisor debe transmitir a una tasa media menor que una trama por tiempo Δt. Es más, si suponemos que en el hardware del receptor no se realiza de manera automática el almacenamiento en el búfer y el encolamiento, el emisor nunca debe transmitir una trama nueva hasta que la vieja haya sido obtenida por *from_physical_layer*, para que lo nuevo no sobrescriba lo antiguo.

En ciertas circunstancias restringidas (por ejemplo, transmisión síncrona y una capa de enlace de datos receptora dedicada por completo a procesar la línea de entrada única), el emisor podría introducir simplemente un retardo en el protocolo 1 y así reducir su velocidad lo suficiente para evitar que se sature el receptor. Sin embargo, es más común que la capa de enlace de datos tenga varias líneas a las cuales atender, y el intervalo de tiempo entre la llegada de una trama y su procesamiento puede variar en forma considerable. Si los diseñadores de la red pueden calcular el comportamiento de peor caso del receptor, podrán programar al emisor para que transmita con tanta lentitud que, aun si cada trama sufre el retardo máximo, no haya desbordamientos. El problema con este método es que es demasiado conservador. Conduce a un aprovechamiento del ancho de banda muy por debajo del óptimo, a menos que el mejor caso y el peor sean iguales (es decir, la variación en el tiempo de reacción de la capa de enlace de datos sea pequeña).

Una solución más general para este dilema es hacer que el receptor proporcione retroalimentación al emisor. Tras haber pasado un paquete a su capa de red, el receptor regresa al emisor una pequeña trama ficticia que, de hecho, autoriza al emisor para transmitir la siguiente trama. Tras haber enviado una trama, el protocolo exige que el emisor espere hasta que llegue la pequeña trama ficticia (es decir, la confirmación de recepción). Utilizar la retroalimentación del receptor para indicar al emisor cuándo puede enviar más datos es un ejemplo del control de flujo que se mencionó anteriormente.

Los protocolos en los que el emisor envía una trama y luego espera una confirmación de recepción antes de continuar se denominan de **parada y espera.** En la figura 3-11 se da un ejemplo de un protocolo símplex de parada y espera.

Aunque el tráfico de datos en este ejemplo es símplex, y va sólo desde el emisor al receptor, las tramas viajan en ambas direcciones. En consecuencia, el canal de comunicación entre las dos capas de enlace de datos necesita tener capacidad de transferencia de información bidireccional. Sin embargo, este protocolo implica una alternancia estricta de flujo: primero el emisor envía una trama, después el receptor envía una trama, después el emisor envía otra trama, después el receptor envía otra, y así sucesivamente. Aquí sería suficiente un canal físico semidúplex.

```
/* El protocolo 2 (parada y espera) también contempla un flujo unidireccional
de datos del emisor al receptor. Se da por hecho nuevamente que el canal de
comunicación está libre de errores, como en el protocolo 1. Sin embargo, esta
vez el receptor tiene capacidad finita de búfer y capacidad finita de
procesamiento, por lo que el protocolo debe evitar de manera explícita que
el emisor sature al receptor con datos a mayor velocidad de la que puede
manejar. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender2(void)
{
  frame s;                          /* búfer para una trama de salida */
  packet buffer;                    /* búfer para un paquete de salida */
  event_type event;                 /* frame_arrival es la única posibilidad */

  while (true) {
     from_network_layer(&buffer);   /* consigue algo que enviar */
     s.info = buffer;               /* lo copia en s para transmisión */
     to_physical_layer(&s);         /* adiós a la pequeña trama */
     wait_for_event(&event);        /* no procede hasta que recibe la señal de
                                       continuación */
  }
}

void receiver2(void)
{
  frame r, s;                       /* búferes para las tramas */
  event_type event;                 /* frame_arrival es la única posibilidad */
  while (true) {
     wait_for_event(&event);        /* la única posibilidad es frame_arrival */
     from_physical_layer(&r);       /* obtiene la trama entrante */
     to_network_layer(&r.info);     /* pasa los datos a la capa de red */
     to_physical_layer(&s);         /* envía una trama ficticia para informar
                                       al emisor */
  }
}
```

Figura 3-11. Protocolo símplex de parada y espera.

Al igual que en el protocolo 1, el emisor comienza obteniendo un paquete de la capa de red, usándolo para construir una trama y enviarla a su destino. Sólo que ahora, a diferencia del protocolo 1, el emisor debe esperar hasta que llegue una trama de confirmación de recepción antes de reiniciar el ciclo y obtener el siguiente paquete de la capa de red. La capa de enlace de datos emisora no necesita inspeccionar la trama entrante, ya que sólo hay una posibilidad: la trama siempre es de confirmación de recepción.

La única diferencia entre *receiver1* y *receiver2* es que, tras entregar un paquete a la capa de red, *receiver2* regresa al emisor una trama de confirmación de recepción antes de entrar nuevamente en el ciclo de espera. Puesto que sólo es importante la llegada de la trama en el emisor, no su contenido, el receptor no necesita poner ninguna información específica en él.

3.3.3 Protocolo símplex para un canal con ruido

Ahora consideremos la situación normal de un canal de comunicación que comete errores. Las tramas pueden llegar dañadas o perderse por completo. Sin embargo, suponemos que si una trama se daña en tránsito, el hardware del receptor detectará esto cuando calcule la suma de verificación. Si la trama está dañada de tal manera que pese a ello la suma de verificación sea correcta, un caso excesivamente improbable, este protocolo (y todos los demás) puede fallar (es decir, entregar un paquete incorrecto a la capa de red).

A primera vista puede parecer que funcionaría una variación del protocolo 2: agregar un temporizador. El emisor podría enviar una trama, pero el receptor sólo enviaría una trama de confirmación de recepción si los datos llegaran correctamente. Si llegara una trama dañada al receptor, se desecharía. Poco después, el temporizador del emisor expiraría y se enviaría la trama de nuevo. Este proceso se repetiría hasta que la trama por fin llegara intacta.

El esquema anterior tiene un defecto mortal. Medite el problema e intente descubrir lo que podría fallar antes de continuar leyendo.

Para ver lo que puede resultar mal, recuerde que la capa de enlace de datos debe proporcionar una comunicación transparente y libre de errores entre los procesos de las capas de red. La capa de red de la máquina *A* pasa una serie de paquetes a la capa de enlace de datos, que debe asegurar que se entregue una serie de paquetes idéntica a la capa de red de la máquina *B* a través de su capa de enlace de datos. En particular, la capa de red en *B* no tiene manera de saber si el paquete se ha perdido o se ha duplicado, por lo que la capa de enlace de datos debe garantizar que ninguna combinación de errores de transmisión, por improbables que sean, pueda causar la entrega de un paquete duplicado a la capa de red.

Considere el siguiente escenario:

1. La capa de red de *A* entrega el paquete 1 a su capa de enlace de datos. El paquete se recibe correctamente en *B* y se pasa a la capa de red de *B*. *B* regresa a *A* una trama de confirmación de recepción.
2. La trama de confirmación de recepción se pierde por completo. Nunca llega. La vida sería mucho más sencilla si el canal sólo alterara o perdiera tramas de datos y no tramas de control, pero desgraciadamente el canal no hace distinciones.
3. El temporizador de la capa de enlace de datos de *A* expira en algún momento. Al no haber recibido una confirmación de recepción, supone (incorrectamente) que su trama de datos se ha perdido o dañado, y envía otra vez la trama que contiene el paquete 1.

4. La trama duplicada también llega bien a la capa de enlace de datos de *B* y de ahí se pasa de manera inadvertida a la capa de red. Si *A* está enviando un archivo a *B*, parte del archivo se duplicará (es decir, la copia del archivo reconstruida por *B* será incorrecta y el error no se habrá detectado). En otras palabras, el protocolo fallará.

Es claro que lo que se necesita es alguna manera de que el receptor sea capaz de distinguir entre una trama que está viendo por primera vez y una retransmisión. La forma evidente de lograr esto es hacer que el emisor ponga un número de secuencia en el encabezado de cada trama que envía. A continuación, el receptor puede examinar el número de secuencia de cada trama que llega para ver si es una trama nueva o un duplicado que debe descartarse.

Dado que es deseable que el encabezado de las tramas sea pequeño, surge la pregunta: ¿cuál es la cantidad mínima de bits necesarios para el número de secuencia? La única ambigüedad de este protocolo es entre una trama, *m*, y su sucesor directo, $m + 1$. Si la trama *m* se pierde o se daña, el receptor no confirmará su recepción y el emisor seguirá tratando de enviarla. Una vez que la trama se recibe correctamente, el receptor regresa una confirmación de recepción al emisor. Es aquí donde surge el problema potencial. Dependiendo de si el emisor recibe correctamente la trama de confirmación de recepción, tratará de enviar *m* o $m + 1$.

El evento que indica al emisor que puede enviar $m + 2$ es la llegada de una confirmación de recepción de $m + 1$. Pero esto implica que *m* se recibió de manera correcta, y además que su confirmación de recepción fue recibida correctamente por el emisor (de otra manera, el emisor no habría comenzado con $m + 1$, y mucho menos con $m + 2$). Como consecuencia, la única ambigüedad es entre una trama y su antecesor o sucesor inmediatos, no entre el antecesor y el sucesor mismos.

Por lo tanto, basta con un número de secuencia de 1 bit (0 o 1). En cada instante, el receptor espera un número de secuencia en particular. Cualquier trama de entrada que contenga un número de secuencia equivocado se rechaza como duplicado. Cuando llega una trama que contiene el número de secuencia correcto, se acepta y se pasa a la capa de red, y el número de secuencia esperado se incrementa módulo 2 (es decir, 0 se vuelve 1 y 1 se vuelve 0).

En la figura 3-12 se muestra un ejemplo de este tipo de protocolo. Los protocolos en los que el emisor espera una confirmación de recepción positiva antes de avanzar al siguiente elemento de datos suelen llamarse **PAR (Confirmación de Recepción Positiva con Retransmisión)** o **ARQ (Solicitud Automática de Repetición).** Al igual que el protocolo 2, éste también transmite datos en una sola dirección.

El protocolo 3 difiere de sus antecesores en que tanto el emisor como el receptor tienen una variable cuyo valor se recuerda mientras la capa de enlace de datos está en estado de espera. El emisor recuerda el número de secuencia de la siguiente trama a enviar en *next_frame_to_send*; el receptor recuerda el número de secuencia de la siguiente trama esperada en *frame_expected*. Cada protocolo tiene una fase de inicialización corta antes de entrar en el ciclo infinito.

```
/* El protocolo 3 (par) permite el flujo unidireccional de datos por un canal no con-
fiable. */

#define MAX_SEQ 1                                   /* debe ser 1 para el protocolo 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;                        /* número de secuencia de la siguiente
                                                       trama de salida */
  frame s;                                          /* variable de trabajo */
  packet buffer;                                    /* búfer para un paquete de salida */
  event_type event;

  next_frame_to_send = 0;                           /* inicializa números de secuencia de
                                                       salida */
  from_network_layer(&buffer);                      /* obtiene el primer paquete */
  while (true){
     s.info = buffer;                               /* construye una trama para transmisión */
     s.seq = next_frame_to_send;                    /* inserta un número de secuencia en la
                                                       trama */
     to_physical_layer(&s);                         /* la envía a su destino */
     start_timer(s.seq);                            /* si la respuesta tarda mucho, expira el
                                                       temporizador */
     wait_for_event(&event);                        /* frame_arrival, cksum_err, timeout */
     if (event == frame_arrival){
           from_physical_layer(&s);                 /* obtiene la confirmación de recepción */
           if (s.ack == next_frame_to_send){
                 stop_timer(s.ack);                 /* desactiva el temporizador */
                 from_network_layer(&buffer);       /* obtiene siguiente a enviar */
                 inc(next_frame_to_send);           /* invierte next_frame_to_send */
           }
     }
  }
}

void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true){
     wait_for_event(&event);                        /* posibilidades: frame_arrival, cksum_err */
     if (event == frame_arrival){                   /* ha llegado una trama válida. */
           from_physical_layer(&r);                 /* obtiene la trama recién llegada */
           if (r.seq == frame_expected){            /* esto es lo que hemos estado esperando. */
                 to_network_layer(&r.info);         /* pasa los datos a la capa de red */
                 inc(frame_expected);               /* para la próxima se espera el otro número
                                                       de secuencia */
           }
           s.ack = 1 - frame_expected;              /* indica la trama cuya recepción se está
                                                       confirmando */
           to_physical_layer(&s);                   /* envía confirmación de recepción */
     }
  }
}
```

Figura 3-12. Protocolo de confirmación de recepción positiva con retransmisión

Tras transmitir una trama, el emisor arranca el temporizador. Si éste ya se estaba ejecutando, se restablece para conceder otro intervalo completo de temporización. Dicho intervalo debe escogerse de modo que haya suficiente tiempo para que la trama llegue al receptor, éste la procese en el peor caso y la confirmación de recepción se regrese al emisor. Sólo cuando ha transcurrido ese intervalo de tiempo se puede suponer con seguridad que se ha perdido la trama transmitida o su confirmación de recepción, y que se debe enviar un duplicado. Si el intervalo establecido es muy pequeño, el emisor transmitirá tramas innecesarias. Si bien estas tramas adicionales no afectarán la corrección del protocolo, sí dañarán el rendimiento.

Tras transmitir una trama y arrancar el temporizador, el emisor espera que ocurra algo interesante. Hay tres posibilidades: llega una trama de confirmación de recepción sin daño, llega una trama de confirmación de recepción dañada o expira el temporizador. Si recibe una confirmación de recepción válida, el emisor obtiene el siguiente paquete de la capa de red y lo coloca en el búfer, sobrescribiendo el paquete previo. También avanza el número de secuencia. Si llega una trama dañada o no llega ninguna, ni el búfer ni el número de secuencia cambia, con el fin de que se pueda enviar un duplicado.

Cuando llega una trama válida al receptor, su número de secuencia se verifica para saber si es un duplicado. Si no lo es, se acepta, se pasa a la capa de red y se genera una confirmación de recepción. Los duplicados y las tramas dañadas no se pasan a la capa de red.

3.4 PROTOCOLOS DE VENTANA CORREDIZA

En los protocolos previos, las tramas de datos se transmiten en una sola dirección. En la mayoría de las situaciones prácticas hay necesidad de transmitir datos en ambas direcciones. Una manera de lograr una transmisión de datos dúplex total es tener dos canales de comunicación separados y utilizar cada uno para tráfico de datos símplex (en diferentes direcciones). Si se hace esto, tenemos dos circuitos físicos separados, cada uno con un canal "de ida" (para datos) y un canal "de retorno" (para confirmaciones de recepción). En ambos casos, el ancho de banda del canal usado para confirmaciones de recepción se desperdicia casi por completo. En efecto, el usuario está pagando dos circuitos, pero sólo usa la capacidad de uno.

Una mejor idea es utilizar el mismo circuito para datos en ambas direcciones. Después de todo, en los protocolos 2 y 3 ya se usaba para transmitir tramas en ambos sentidos, y el canal de retorno tiene la misma capacidad que el canal de ida. En este modelo, las tramas de datos de *A* a *B* se mezclan con las tramas de confirmación de recepción de *A* a *B*. Analizando el campo de tipo (*kind*) en el encabezado de una trama de entrada, el receptor puede saber si la trama es de datos o de confirmación de recepción.

Aunque el entrelazado de datos y de tramas de control en el mismo circuito es una mejora respecto al uso de dos circuitos físico separados, se puede lograr otra mejora. Cuando llega una trama de datos, en lugar de enviar inmediatamente una trama de control independiente, el receptor se aguanta y espera hasta que la capa de red le pasa el siguiente paquete. La confirmación de recepción se anexa a la trama de datos de salida (usando el campo *ack* del encabezado de la trama). En efecto, la confirmación de recepción viaja gratuitamente en la siguiente trama de datos de salida.

La técnica de retardar temporalmente las confirmaciones de recepción para que puedan viajar en la siguiente trama de datos de salida se conoce como **superposición** (*piggybacking*).

La ventaja principal de usar la superposición en lugar de tener tramas de confirmación de recepción independientes es un mejor aprovechamiento del ancho de banda disponible del canal. El campo *ack* del encabezado de la trama ocupa sólo unos cuantos bits, mientras que una trama aparte requeriría de un encabezado, la confirmación de recepción y una suma de verificación. Además, el envío de menos tramas implica menos interrupciones de "ha llegado trama" y tal vez menos segmentos de búfer en el receptor, dependiendo de la manera en que esté organizado el software del receptor. En el siguiente protocolo que examinaremos, el campo de superposición ocupa sólo 1 bit en el encabezado de la trama y pocas veces ocupa más de algunos bits.

Sin embargo, la superposición introduce una complicación inexistente en las confirmaciones de recepción independientes. ¿Cuánto tiempo debe esperar la capa de enlace de datos un paquete al cual superponer la confirmación de recepción? Si la capa de enlace de datos espera más tiempo del que tarda en terminar el temporizador del emisor, la trama será retransmitida, frustrando el propósito de enviar confirmaciones de recepción. Si la capa de enlace de datos fuera un oráculo y pudiera predecir el futuro, sabría cuándo se recibiría el siguiente paquete de la capa de red y podría decidir esperarlo o enviar de inmediato una confirmación de recepción independiente, dependiendo del tiempo de espera proyectado. Por supuesto, la capa de enlace de datos no puede predecir el futuro, por lo que debe recurrir a algún esquema particular para el caso, como esperar un número fijo de milisegundos. Si llega rápidamente un nuevo paquete, la confirmación de recepción se superpone a él; de otra manera, si no ha llegado ningún paquete nuevo al final de este periodo, la capa de enlace de datos manda una trama de confirmación de recepción independiente.

Los siguientes tres protocolos son bidireccionales y pertenecen a una clase llamada protocolos de **ventana corrediza.** Los tres difieren entre ellos en la eficiencia, complejidad y requerimientos de búfer, como se analizará más adelante. En ellos, al igual que en todos los protocolos de ventana corrediza, cada trama de salida contiene un número de secuencia, que va desde 0 hasta algún número máximo. Por lo general, éste es $2^n - 1$, por lo que el número de secuencia encaja perfectamente en un campo de n bits. El protocolo de ventana corrediza de parada y espera utiliza $n = 1$, y restringe los números de secuencia de 0 y 1, pero las versiones más refinadas pueden utilizar un n arbitrario.

La esencia de todos los protocolos de ventana corrediza es que, en cualquier instante, el emisor mantiene un grupo de números de secuencia que corresponde a las tramas que tiene permitido enviar. Se dice que estas tramas caen dentro de la **ventana emisora.** De manera semejante, el receptor mantiene una **ventana receptora** correspondiente al grupo de tramas que tiene permitido aceptar. La ventana del emisor y la del receptor no necesitan tener los mismos límites inferior y superior, ni siquiera el mismo tamaño. En algunos protocolos las ventanas son de tamaño fijo, pero en otros pueden crecer y disminuir a medida que se envían y reciben las tramas.

Aunque estos protocolos dan a la capa de enlace de datos mayor libertad en cuanto al orden en que puede enviar y recibir tramas, hemos conservado decididamente el requisito de que el protocolo debe entregar los paquetes a la capa de red del destino en el mismo orden en que se pasaron a la capa de enlace de datos de la máquina emisora. Tampoco hemos cambiado el requisito de que

el canal físico de comunicación es "de tipo alambre", es decir, que debe entregar todas las tramas en el orden en que fueron enviadas.

Los números de secuencia en la ventana del emisor representan tramas enviadas, o que pueden ser enviadas, pero cuya recepción aún no se ha confirmado. Cuando llega un paquete nuevo de la capa de red, se le da el siguiente número secuencial mayor, y el extremo superior de la ventana avanza en uno. Al llegar una confirmación de recepción, el extremo inferior avanza en uno. De esta manera, la ventana mantiene continuamente una lista de tramas sin confirmación de recepción. En la figura 3-13 se muestra un ejemplo.

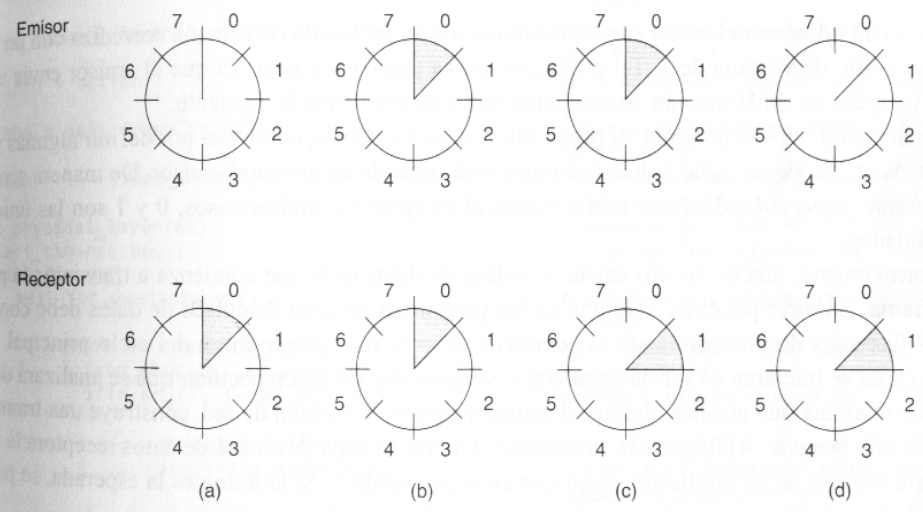


Figura 3-13. Ventana corrediza de tamaño 1, con un número de secuencia de 3 bits. (a) Al inicio. (b) Tras la transmisión de la primera trama. (c) Tras la recepción de la primera trama. (d) Tras recibir la primera confirmación de recepción.

Dado que las tramas que están en la ventana del emisor pueden perderse o dañarse en tránsito, el emisor debe mantener todas estas tramas en su memoria para su posible retransmisión. Por lo tanto, si el tamaño máximo de la ventana es *n*, el emisor necesita *n* búferes para contener las tramas sin confirmación de recepción. Si la ventana llega a crecer a su tamaño máximo, la capa de enlace de datos emisora deberá hacer que la capa de red se detenga hasta que se libere otro búfer.

La ventana de la capa de enlace de datos receptora corresponde a las tramas que puede aceptar. Toda trama que caiga fuera de la ventana se descartará sin comentarios. Cuando se recibe la trama cuyo número de secuencia es igual al extremo inferior de la ventana, se pasa a la capa de red, se genera una confirmación de recepción y se avanza la ventana en uno. A diferencia de la ventana del emisor, la ventana del receptor conserva siempre el mismo tamaño inicial. Note que

un tamaño de ventana de 1 significa que la capa de enlace de datos sólo acepta tramas en orden, pero con ventanas más grandes esto no es así. La capa de red, en contraste, siempre recibe los datos en el orden correcto, sin importar el tamaño de la ventana de la capa de enlace de datos.

En la figura 3-13 se muestra un ejemplo con un tamaño máximo de ventana de 1. Inicialmente no hay tramas pendientes, por lo que los extremos de la ventana del emisor son iguales, pero a medida que pasa el tiempo, la situación progresa como se muestra.

3.4.1 Un protocolo de ventana corrediza de un bit

Antes de lidiar con el caso general, examinemos un protocolo de ventana corrediza con un tamaño máximo de ventana de 1. Tal protocolo utiliza parada y espera, ya que el emisor envía una trama y espera su confirmación de recepción antes de transmitir la siguiente.

En la figura 3-14 se presenta tal protocolo. Como los demás, comienza por definir algunas variables. *Next_frame_to_send* indica qué trama está tratando de enviar el emisor. De manera semejante, *frame_expected* indica qué trama espera el receptor. En ambos casos, 0 y 1 son las únicas posibilidades.

Normalmente, una de las dos capas de enlace de datos es la que comienza a transmitir la primera trama. En otras palabras, sólo uno de los programas de capa de enlace de datos debe contener las llamadas de procedimiento *to_physical_layer* y *start_timer* fuera del ciclo principal. Si ambas capas se iniciaran en forma simultánea, surgiría una situación peculiar que se analizará después. La máquina que arranca obtiene el primer paquete de su capa de red, construye una trama a partir de él y la envía. Al llegar esta (o cualquier) trama, la capa de enlace de datos receptora la revisa para saber si es un duplicado, igual que en el protocolo 3. Si la trama es la esperada, se pasa a la capa de red y la ventana del receptor se recorre hacia arriba.

El campo de confirmación de recepción contiene el número de la última trama recibida sin error. Si este número concuerda con el de secuencia de la trama que está tratando de enviar el emisor, éste sabe que ha terminado con la trama almacenada en el *búfer* y que puede obtener el siguiente paquete de su capa de red. Si el número de secuencia no concuerda, debe continuar intentando enviar la misma trama. Por cada trama que se recibe, se regresa una.

Ahora examinemos el protocolo 4 para ver qué tan flexible es ante circunstancias patológicas. Suponga que *A* está tratando de enviar su trama 0 a *B* y que *B* está tratando de enviar su trama 0 a *A*. Suponga que *A* envía una trama a *B*, pero que el intervalo de temporización de *A* es un poco corto. En consecuencia, *A* podría terminar su temporización repetidamente, enviando una serie de tramas idénticas, todas con $seq = 0$ y $ack = 1$.

Al llegar la primera trama válida a *B*, es aceptada y *frame_expected* se establece en 1. Todas las tramas subsiguientes serán rechazadas, pues *B* ahora espera tramas con el número de secuencia 1, no 0. Además, dado que los duplicados tienen $ack = 1$ y *B* aún está esperando una confirmación de recepción de 0, *B* no extraerá un nuevo paquete de su capa de red.

```
/* El protocolo 4 (de ventana corrediza) es bidireccional. */
#define MAX_SEQ 1                                   /* debe ser 1 para el protocolo 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
 seq_nr next_frame_to_send;                         /* sólo 0 o 1 */
 seq_nr frame_expected;                             /* sólo 0 o 1 */
 frame r, s;                                        /* variables de trabajo */
 packet buffer;                                     /* paquete actual que se envía */
 event_type event;
 next_frame_to_send = 0;                            /* siguiente trama del flujo de
                                                       salida */
 frame_expected = 0;                                /* número de trama de llegada
                                                       esperada */
 from_network_layer(&buffer);                       /* obtiene un paquete de la capa de
                                                       red */
 s.info = buffer;                                   /* se prepara para enviar la trama
                                                       inicial */
 s.seq = next_frame_to_send;                        /* inserta el número de secuencia en
                                                       la trama */
 s.ack = 1 - frame_expected;                        /* confirmación de recepción
                                                       superpuesta */
 to_physical_layer(&s);                             /* transmite la trama */
 start_timer(s.seq);                                /* inicia el temporizador */
 while (true){
    wait_for_event(&event);                         /* frame_arrival, cksum_err o
                                                       timeout */
    if (event == frame_arrival){                    /* ha llegado una trama sin daño. */
          from_physical_layer(&r);                  /* lo obtiene */
          if(r.seq == frame_expected) {             /* maneja flujo de tramas de
                                                       entrada. */
                to_network_layer(&r.info);          /* pasa el paquete a la capa de red */
                inc(frame_expected);                /* invierte el siguiente número de
                                                       secuencia esperado */
          }
          if(r.ack == next_frame_to_send){          /* maneja flujo de tramas de
                                                       salida. */
                stop_timer(r.ack);                  /* desactiva el temporizador */
                from_network_layer(&buffer);        /* obtiene paquete nuevo de la capa
                                                       de red */
                inc(next_frame_to_send);            /* invierte el número de secuencia
                                                       del emisor */
          }
    }
    s.info = buffer;                                /* construye trama de salida */
    s.seq = next_frame_to_send;                     /* le introduce el número de
                                                       secuencia */
    s.ack = 1 - frame_expected;                     /* número de secuencia de la última
                                                       trama recibida */
    to_physical_layer(&s);                          /* transmite una trama */
    start_timer(s.seq);                             /* inicia el temporizador */
 }
}
```

Figura 3-14. Protocolo de ventana corrediza de 1 bit.

Cada vez que llega un duplicado rechazado, *B* envía a *A* una trama que contiene $seq = 0$ y $ack = 0$. Tarde o temprano una de éstas llegará correctamente a *A*, haciendo que *A* comience a enviar el siguiente paquete. Ninguna combinación de tramas perdidas o expiración de temporizadores puede hacer que el protocolo entregue paquetes duplicados a cualquiera de las capas de red, ni que omita un paquete, ni que entre en un bloqueo irreversible.

Sin embargo, si ambos lados envían de manera simultánea un paquete inicial, surge una situación peculiar. En la figura 3-15 se muestra este problema de sincronización. En la parte (a) se muestra la operación normal del protocolo. En (b) se ilustra la peculiaridad. Si *B* espera la primera trama de *A* antes de enviar la suya, la secuencia es como se muestra en (a), y todas las tramas son aceptadas. Sin embargo, si *A* y *B* inician la comunicación simultáneamente, se cruzan sus primeras tramas y las capas de enlace de datos entran en la situación (b). En (a) cada trama que llega trae un paquete nuevo para la capa de red; no hay duplicados. En (b) la mitad de las tramas contienen duplicados, aun cuando no hay errores de transmisión. Pueden ocurrir situaciones similares como resultado de la expiración prematura de temporizadores, aun cuando un lado comience evidentemente primero. De hecho, si ocurren varias expiraciones prematuras de temporizadores las tramas podrían enviarse tres o más veces.

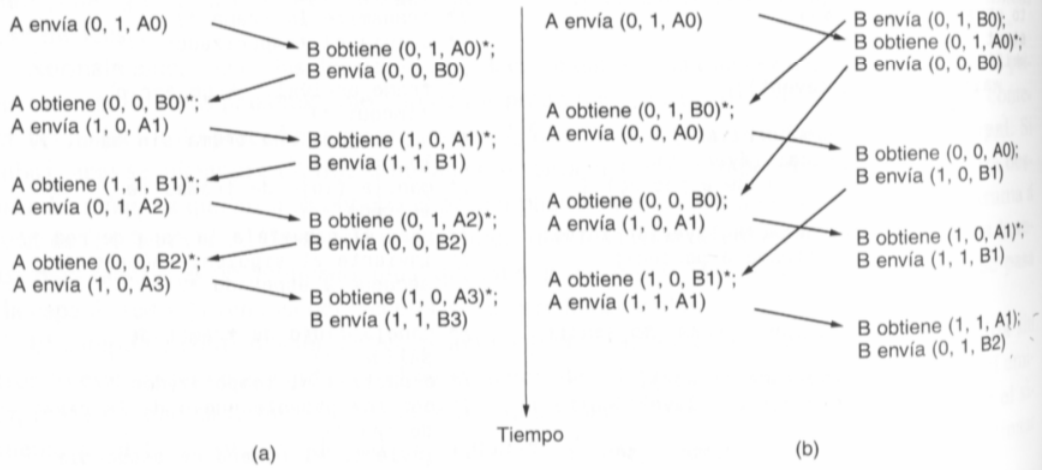


Figura 3-15. Dos escenarios para el protocolo 4. (a) Caso normal. (b) Caso anormal. La notación es (secuencia, confirmación de recepción, número de paquete). Un asterisco indica el lugar en que una capa de red acepta un paquete.

3.4.2 Protocolo que usa retroceso N

Hasta ahora hemos supuesto que el tiempo de transmisión requerido para que una trama llegue al receptor más el necesario para que la confirmación de recepción regrese es insignificante. A veces esta suposición es totalmente falsa. En estas situaciones el tiempo de viaje de ida y vuelta prolongado puede tener implicaciones importantes para la eficiencia del aprovechamiento del ancho

de banda. Por ejemplo, considere un canal de satélite de 50 kbps con un retardo de propagación de ida y vuelta de 500 mseg. Imagine que intentamos utilizar el protocolo 4 para enviar tramas de 1000 bits por medio del satélite. El emisor empieza a enviar la primera trama en $t = 0$. En $t =$ 20 mseg la trama ha sido enviada por completo. En las mejores circunstancias (sin esperas en el receptor y una trama de confirmación de recepción corta), no es sino hasta $t = 270$ mseg que la trama ha llegado por completo al receptor, y no es sino hasta $t = 520$ mseg que ha llegado la confirmación de recepción de regreso al emisor. Esto implica que el emisor estuvo bloqueado durante el 500/520 = 96% del tiempo. En otras palabras, sólo se usó el 4% del ancho de banda disponible. Queda claro que la combinación de un tiempo de tránsito grande, un ancho de banda alto y una longitud de tramas corta es desastrosa para la eficiencia.

El problema antes descrito puede verse como una consecuencia de la regla que requiere que el emisor espere una confirmación de recepción antes de enviar otra trama. Si relajamos esa restricción, se puede lograr una mejor eficiencia. Básicamente la solución está en permitir que el emisor envíe hasta w tramas antes de bloquearse, en lugar de sólo 1. Con una selección adecuada de w, el emisor podrá transmitir tramas continuamente durante un tiempo igual al tiempo de tránsito de ida y vuelta sin llenar la ventana. En el ejemplo anterior, w debe ser de cuando menos 26. El emisor comienza por enviar la trama 0, como antes. Para cuando ha terminado de enviar 26 tramas, en $t = 520$, llega la confirmación de recepción de la trama 0. A partir de entonces, las confirmaciones de recepción llegarán cada 20 mseg, por lo que el emisor siempre tendrá permiso de continuar justo cuando lo necesita. En todo momento hay 25 o 26 tramas pendientes de confirmación de recepción. Dicho de otra manera, el tamaño máximo de la ventana del emisor es de 26.

La necesidad de una ventana grande en el lado emisor se presenta cuando el producto del ancho de banda por el retardo del viaje de ida y vuelta es grande. Si el ancho de banda es alto, incluso para un retardo moderado, el emisor agotará su ventana rápidamente a menos que tenga una ventana grande. Si el retardo es grande (por ejemplo, en un canal de satélite geoestacionario), el emisor agotará su ventana incluso con un ancho de banda moderado. El producto de estos dos factores indica básicamente cuál es la capacidad del canal, y el emisor necesita la capacidad de llenarlo sin detenerse para poder funcionar con una eficiencia máxima.

Esta técnica se conoce como **canalización.** Si la capacidad del canal es de b bits/seg, el tamaño de la trama de l bits y el tiempo de propagación de ida y vuelta de R segundos, el tiempo requerido para transmitir una sola trama es de l/b segundos. Una vez que ha sido enviado el último bit de una trama de datos, hay un retardo de $R/2$ antes de que llegue ese bit al receptor y un retardo de cuando menos $R/2$ para que la confirmación de recepción llegue de regreso, lo que da un retardo total de R. En parada y espera, la línea está ocupada durante l/b e inactiva durante R, dando

$$\text{una utilización de la línea de} = l/(l + bR)$$

Si $l < bR$, la eficiencia será menor que 50%. Ya que siempre hay un retardo diferente de cero para que la confirmación de recepción se propague de regreso, en principio la canalización puede servir para mantener ocupada la línea durante este intervalo, pero si el intervalo es pequeño, la complejidad adicional no justifica el esfuerzo.

El envío de tramas en canalización por un canal de comunicación inestable presenta problemas serios. Primero, ¿qué ocurre si una trama a la mitad de una serie larga se daña o pierde? Llegarán grandes cantidades de tramas sucesivas al receptor antes de que el emisor se entere de que algo anda mal. Cuando llega una trama dañada al receptor, obviamente debe descartarse, pero, ¿qué debe hacerse con las tramas correctas que le siguen? Recuerde que la capa de enlace de datos receptora está obligada a entregar paquetes a la capa de red en secuencia. En la figura 3-16 se muestran los efectos de la canalización en la recuperación de un error. A continuación lo analizaremos en detalle.

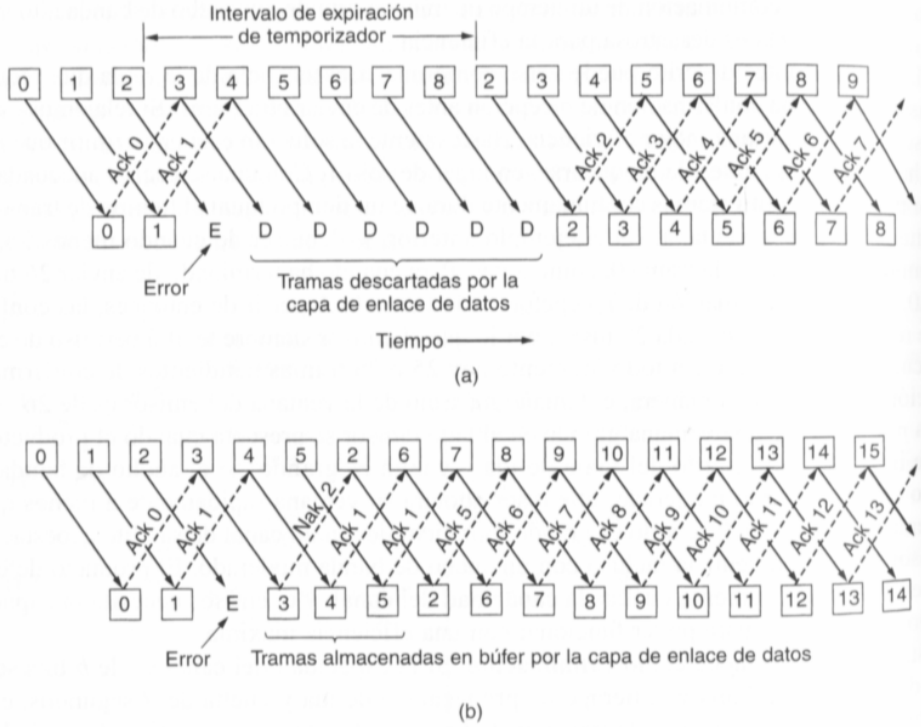


Figura 3-16. Canalización y recuperación de un error. (a) Efecto de un error cuando el tamaño de la ventana del receptor es de 1. (b) Efecto de un error cuando el tamaño de la ventana del receptor es grande.

Hay dos métodos básicos para manejar los errores durante la canalización. Una manera, llamada **retroceso n,** es que el receptor simplemente descarte todas las tramas subsecuentes, sin enviar confirmaciones de recepción para las tramas descartadas. Esta estrategia corresponde a una ventana de recepción de tamaño 1. En otras palabras, la capa de enlace de datos se niega a aceptar cualquier trama excepto la siguiente que debe entregar a la capa de red. Si la ventana del emisor se llena antes de terminar el temporizador, el canal comenzará a vaciarse. En algún momento, el emisor terminará de esperar y retransmitirá en orden todas las tramas cuya recepción aún no se

haya confirmado, comenzando por la dañada o perdida. Esta estrategia puede desperdiciar bastante ancho de banda si la tasa de errores es alta.

En la figura 3-16(a) se muestra el retroceso n en el caso en que la ventana del receptor es grande. Las tramas 0 y 1 se reciben y confirman de manera correcta. Sin embargo, la trama 2 se daña o pierde. El emisor, no consciente de este problema, continúa enviando tramas hasta que expira el temporizador para la trama 2. Después retrocede a la trama 2 y comienza con ella, enviando 2, 3, 4, etcétera, nuevamente.

La otra estrategia general para el manejo de errores cuando las tramas se colocan en canalizaciones se conoce como **repetición selectiva.** Cuando se utiliza, se descarta una trama dañada recibida, pero las tramas en buen estado recibidas después de ésa se almacenan en el búfer. Cuando el emisor termina, sólo la última trama sin confirmación se retransmite. Si la trama llega correctamente, el receptor puede entregar a la capa de red, en secuencia, todas las tramas que ha almacenado en el búfer. La repetición selectiva con frecuencia se combina con el hecho de que el receptor envíe una confirmación de recepción negativa (NAK) cuando detecta un error, por ejemplo, cuando recibe un error de suma de verificación o una trama en desorden. Las confirmaciones de recepción negativas estimulan la retransmisión antes de que el temporizador correspondiente expire y, por lo tanto, mejoran el rendimiento.

En la figura 3-16(b), las tramas 0 y 1 se vuelven a recibir y confirmar correctamente y la trama 2 se pierde. Cuando la trama 3 llega al receptor, su capa de enlace de datos observa que falta una trama, por lo que regresa una NAK para la trama 2 pero almacena la trama 3. Cuando las tramas 4 y 5 llegan, también son almacenadas por la capa de enlace de datos en lugar de pasarse a la capa de red. En algún momento, la NAK 2 llega al emisor, que inmediatamente reenvía la trama 2. Cuando llega, la capa de enlace de datos ahora tiene 2, 3, 4 y 5 y ya las puede pasar a la capa de red en el orden correcto. También puede confirmar la recepción de todas las tramas hasta, e incluyendo, la 5, como se muestra en la figura. Si la NAK se perdiera, en algún momento el temporizador del emisor expirará para la trama 2 y la enviará (sólo a ella), pero eso puede tardar un poco más. En efecto, la NAK acelera la retransmisión de una trama específica.

La repetición selectiva corresponde a una ventana del receptor mayor que 1. Cualquier trama dentro de la ventana puede ser aceptada y mantenida en el búfer hasta que todas las que le preceden hayan sido pasadas a la capa de red. Esta estrategia puede requerir cantidades grandes de memoria en la capa de enlace de datos si la ventana es grande.

Estas dos estrategias alternativas son intercambios entre el ancho de banda y el espacio de búfer en la capa de enlace de datos. Dependiendo de qué recurso sea más valioso, se puede utilizar uno o el otro. En la figura 3-17 se muestra un protocolo de canalización en el que la capa de enlace de datos receptora sólo acepta tramas en orden; las tramas siguientes a un error son descartadas. En este protocolo hemos omitido por primera vez el supuesto de que la capa de red siempre tiene un suministro infinito de paquetes que enviar. Cuando la capa de red tiene un paquete para enviar, puede causar la ocurrencia de un evento *network_layer_ready*. Sin embargo, para poder cumplir la regla de control de flujo de que no debe haber más de *MAX_SEQ* tramas sin confirmación de recepción pendientes en cualquier momento, la capa de enlace de datos debe poder prohibir a la capa de red que la moleste con más trabajo. Los procedimientos de biblioteca *enable_network_layer* y *disable_network_layer* llevan a cabo esta función.

```
/* El protocolo 5 (retroceso n) permite múltiples tramas pendientes. El emisor podría
   enviar hasta MAX_SEQ tramas sin esperar una confirmación de recepción. Además, a
   diferencia de los primeros protocolos, no se supone que la capa de red debe tener
   siempre un paquete nuevo. En vez de ello, la capa de red activa un evento
   network_layer_ready cuando hay un paquete por enviar. */

#define MAX_SEQ 7                       /* debe ser 2^n - 1 */
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Devuelve true si a <=b < c de manera circular, y false en caso contrario. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
      return(true);
    else
      return(false);
}

static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Elabora y envía una trama de datos. */
  frame s;                              /* variable de trabajo */

  s.info = buffer[frame_nr];            /* inserta el paquete en la trama */
  s.seq = frame_nr;                     /* inserta un número de secuencia en la trama */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);      /* ack superpuesta*/
  to_physical_layer(&s);                /* transmite la trama */
  start_timer(frame_nr);                /* inicia la ejecución del temporizador */
}

void protocol5(void)
{
  seq_nr next_frame_to_send;            /* MAX_SEQ > 1; utilizado para flujo de salida */
  seq_nr ack_expected;                  /* la trama más vieja hasta el momento no
                                           confirmada */
  seq_nr frame_expected;                /* siguiente trama esperada en el flujo de
                                           entrada */
  frame r;                              /* variable de trabajo */
  packet buffer[MAX_SEQ + 1];           /* búferes para el flujo de salida */
  seq_nr nbuffered;                     /* # de búferes de salida actualmente en uso */
  seq_nr i;                             /* utilizado para indexar en el arreglo de
                                           búferes */
  event_type event;

  enable_network_layer();               /* permite eventos network_layer_ready */
  ack_expected = 0;                     /* siguiente ack esperado en el flujo
                                           de entrada */
  next_frame_to_send = 0;               /* siguiente trama de salida */
  frame_expected = 0;                   /* número de trama de entrada esperada */
  nbuffered = 0;                        /* en principio no hay paquetes en búfer */
  while (true) {
   wait_for_event(&event);              /* cuatro posibilidades: vea event_type al
                                           principio */
```

```
switch(event) {
case network_layer_ready:          /* la capa de red tiene un paquete para enviar */
     /* Acepta, guarda y transmite una trama nueva. */
     from_network_layer(&buffer[next_frame_to_send]); /* obtiene un paquete nuevo */
     nbuffered = nbuffered + 1;    /* expande la ventana del emisor */
     send_data(next_frame_to_send, frame_expected, buffer); /* transmite la trama */
     inc(next_frame_to_send);      /* avanza el límite superior de la ventana del
                                      emisor */
     break;

case frame_arrival:                /* ha llegado una trama de datos o de control */
     from_physical_layer(&r);      /* obtiene una trama entrante de la capa física */

     if (r.seq == frame_expected) {
          /* Las tramas se aceptan sólo en orden. */
          to_network_layer(&r.info); /* pasa el paquete a la capa de red */
          inc(frame_expected);   /* avanza el límite inferior de la ventana del
                                    receptor */
     }

     /* Ack n implica n - 1, n - 2, etc.  Verificar esto. */
     while (between(ack_expected, r.ack, next_frame_to_send)) {
          /* Maneja la ack superpuesta. */
          nbuffered = nbuffered - 1; /* una trama menos en el búfer */
          stop_timer(ack_expected);  /* la trama llegó intacta; detener el
                                        temporizador */
          inc(ack_expected);     /* reduce la ventana del emisor */
     }
     break;

case cksum_err: break;             /* ignora las tramas erróneas */

case timeout:                      /* problemas; retransmite todas las tramas
                                      pendientes */
     next_frame_to_send = ack_expected; /* inicia aquí la retransmisión */
     for (i = 1; i <= nbuffered; i++) {
          send_data(next_frame_to_send, frame_expected, buffer);/* reenvía la
                                                                   trama 1 */
          inc(next_frame_to_send);   /* se prepara para enviar la siguiente */
     }
}

if (nbuffered < MAX_SEQ)
     enable_network_layer();
else
     disable_network_layer();
}
```

Figura 3-17. Protocolo de ventana corrediza con retroceso n.

Note que pueden como máximo estar pendientes *MAX_SEQ* tramas, y no *MAX_SEQ* + 1 en cualquier momento, aun cuando haya *MAX_SEQ* + 1 números de secuencia diferentes: 0, 1, 2,..., *MAX_SEQ*. Para ver por qué es necesaria esta restricción, considere la siguiente situación con *MAX_SEQ* = 7.

1. El emisor envía las tramas 0 a 7.
2. En algún momento llega al emisor una confirmación de recepción, superpuesta, para la trama 7.
3. El emisor envía otras ocho tramas, nuevamente con los números de secuencia 0 a 7.
4. Ahora llega otra confirmación de recepción, superpuesta, para la trama 7.

La pregunta es: ¿llegaron con éxito las ocho tramas que correspondían al segundo bloque o se perdieron (contando como pérdidas los rechazos siguientes a un error)? En ambos casos el receptor podría estar enviando la trama 7 como confirmación de recepción. El emisor no tiene manera de saberlo. Por esta razón, el número máximo de tramas pendientes debe restringirse a *MAX_SEQ*.

Aunque el protocolo 5 no pone en el búfer las tramas que llegan tras un error, no escapa del problema de los búferes por completo. Dado que un emisor puede tener que retransmitir en un momento futuro todas las tramas no confirmadas, debe retener todas las tramas retransmitidas hasta saber con certeza que han sido aceptadas por el receptor. Al llegar una confirmación de recepción para la trama n, las tramas $n - 1$, $n - 2$, y demás, se confirman de manera automática. Esta propiedad es especialmente importante cuando algunas tramas previas portadoras de confirmaciones de recepción se perdieron o dañaron. Cuando llega una confirmación de recepción, la capa de enlace de datos revisa si se pueden liberar búferes. Si esto es posible (es decir, hay espacio disponible en la ventana), ya puede permitirse que una capa de red previamente bloqueada produzca más eventos *network_layer_ready*.

Para este protocolo damos por hecho que siempre hay tráfico de regreso en el que se pueden superponer confirmaciones de recepción. Si no lo hay, no es posible enviar confirmaciones de recepción. El protocolo 4 no necesita este supuesto debido a que envía una trama cada vez que recibe una trama, incluso si ya ha enviado la trama. En el siguiente protocolo resolveremos el problema del tráfico de una vía de una forma elegante.

Debido a que este protocolo tiene múltiples tramas pendientes, necesita lógicamente múltiples temporizadores, uno por cada trama pendiente. Cada trama termina de temporizar independientemente de todas las demás. Todos estos temporizadores pueden simularse fácilmente en software, usando un solo reloj de hardware que produzca interrupciones periódicas. Las terminaciones de temporización pendientes forman una lista enlazada, en la que cada nodo de la lista indica la cantidad de pulsos de reloj que faltan para que expire el temporizador, el número de la trama temporizada y un apuntador al siguiente nodo.

Como ilustración de una manera de implementar los temporizadores, considere el ejemplo de la figura 3-18(a). Suponga que el reloj pulsa una vez cada 100 mseg. Inicialmente la hora real es 10:00:00.0 y hay tres terminaciones pendientes, a las 10:00:00.5, 10:00:01.3 y 10:00:01.9.

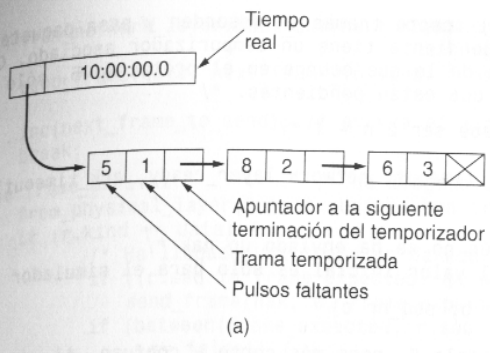


Figura 3-18. Simulación de múltiples temporizadores en software.

Cada vez que pulsa el reloj de hardware, se actualiza el tiempo real y el contador de pulsos a la cabeza de la lista se decrementa. Al llegar a cero el contador de pulsos, se causa una terminación y se retira el nodo de la lista, como se muestra en la figura 3-18(b). Aunque esta organización requiere que la lista se examine al llamar a *start_timer* o a *stop_timer*, no requiere mucho trabajo por pulso. En el protocolo 5 estas dos rutinas tienen un parámetro que indica la trama a temporizar.

3.4.3 Protocolo que utiliza repetición selectiva

El protocolo 5 funciona bien si los errores son poco frecuentes, pero si la línea es mala, se desperdicia mucho ancho de banda en las tramas retransmitidas. Una estrategia alterna para el manejo de errores es permitir que el receptor acepte y coloque en búferes las tramas que siguen a una trama dañada o perdida. Tal protocolo no rechaza tramas simplemente porque se dañó o se perdió una trama anterior.

En este protocolo, tanto el emisor como el receptor mantienen una ventana de números de secuencia aceptables. El tamaño de la ventana del emisor comienza en 0 y crece hasta un máximo predefinido, *MAX_SEQ*. La ventana del receptor, en cambio, siempre es de tamaño fijo e igual a *MAX_SEQ*. El receptor tiene un búfer reservado para cada número de secuencia en su ventana fija. Cada búfer tiene un bit asociado (*arrived*) que indica si el búfer está lleno o vacío. Cuando llega una trama, su número de secuencia es revisado por la función *between* para ver si cae dentro de la ventana. De ser así, y no ha sido recibida aún, se acepta y almacena. Esta acción se lleva a cabo sin importar si la trama contiene el siguiente paquete esperado por la capa de red. Por supuesto, debe mantenerse dentro de la capa de enlace de datos sin entregarse a la capa de red hasta que todas las tramas de número menor hayan sido entregadas a la capa de red en el orden correcto. En la figura 3-19 se presenta un protocolo que usa este algoritmo.

```
/* El protocolo 6 (repetición selectiva) acepta tramas en desorden y pasa paquetes en
   orden a la capa de red. Cada trama pendiente tiene un temporizador asociado. Cuando
   el temporizador expira, a diferencia de lo que ocurre en el protocolo 5, sólo se
   retransmite esa trama, no todas las que están pendientes. */

#define MAX_SEQ 7                       /* debe ser 2^n - 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout}
  event_type;
#include "protocol.h"
boolean no_nak = true;                  /* aún no se ha enviado un nak */
seq_nr oldest_frame = MAX_SEQ + 1; /* el valor inicial es sólo para el simulador */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Parecido a lo que ocurre en el protocolo 5, pero más corto y confuso. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet
  buffer[])
{
/* Construye y envía una trama de datos, ack o nak. */
  frame s;                              /* variable de trabajo */

  s.kind = fk;                          /* kind == datos, ack o nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                     /* sólo tiene importancia para las tramas de datos */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;        /* un nak por trama, por favor */
  to_physical_layer(&s);                /* transmite la trama */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                     /* no es necesario para tramas ack separadas */
}

void protocol6(void)
{
  seq_nr ack_expected;                  /* límite inferior de la ventana del emisor */
  seq_nr next_frame_to_send;            /* límite superior de la ventana del emisor + 1 */
  seq_nr frame_expected;                /* límite inferior de la ventana del receptor */
  seq_nr too_far;                       /* límite superior de la ventana del receptor + 1 */
  int i;                                /* índice en el grupo de búferes */
  frame r;                              /* variable de trabajo*/
  packet out_buf[NR_BUFS];              /* búferes para el flujo de salida */
  packet in_buf[NR_BUFS];               /* búferes para el flujo de entrada */
  boolean arrived[NR_BUFS];             /* mapa de bits de entrada */
  seq_nr nbuffered;                     /* cuántos búferes de salida se utilizan
                                           actualmente */
  event_type event;

  enable_network_layer();               /* inicializar */
  ack_expected = 0;                     /* siguiente ack esperada en el flujo de entrada */
  next_frame_to_send = 0;               /* número de la siguiente trama de salida */
  frame_expected = 0;
  too_far = NR_BUFS;
  nbuffered = 0;                        /* inicialmente no hay paquetes en el búfer */
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;

  while (true) {
   wait_for_event(&event);              /* cinco posibilidades: vea event_type al principio */
   switch(event) {
    case network_layer_ready: /* acepta, guarda y transmite una trama nueva */
         nbuffered = nbuffered + 1;  /* expande la ventana */
```

```
        from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]);/* obtiene un
                                                                    paquete nuevo */
        send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmite la
                                                                    trama */
        inc(next_frame_to_send); /* avanza el límite superior de la ventana */
        break;
    case frame_arrival:          /* ha llegado una trama de datos o de control */
        from_physical_layer(&r);/* obtiene una trama entrante de la capa física */
        if (r.kind == data) {
            /* Ha llegado una trama no dañada. */
            if ((r.seq != frame_expected) && no_nak)
               send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
            if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS]
                == false)) {
                /* Las tramas se podrían aceptar en cualquier orden. */
                arrived[r.seq % NR_BUFS] = true;     /* marca como lleno el búfer */
                in_buf[r.seq % NR_BUFS] = r.info;    /* inserta datos en el búfer */
                while (arrived[frame_expected % NR_BUFS]) {
                    /* Pasa tramas y avanza la ventana. */
                    to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                    no_nak = true;
                    arrived[frame_expected % NR_BUFS] = false;
                    inc(frame_expected);      /* avanza el límite inferior de la
                                                 ventana del receptor */
                    inc(too_far);      /* avanza el límite superior de la
                                          ventana del receptor */
                    start_ack_timer(); /* para saber si es necesaria una ack
                                          separada */
                }
            }
        }
        if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),
           next_frame_to_send))
            send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
        while (between(ack_expected, r.ack, next_frame_to_send)) {
            nbuffered = nbuffered - 1;   /* maneja la ack superpuesta */
            stop_timer(ack_expected % NR_BUFS);  /* la trama llega intacta */
            inc(ack_expected);  /* avanza el límite inferior de la ventana del emisor */
        }
        break;
    case cksum_err:
        if (no_nak) send_frame(nak, 0, frame_expected, out_buf); /* trama dañada */
        break;
    case timeout:
        send_frame(data, oldest_frame, frame_expected, out_buf);  /* hacemos que expire
                                                                     el temporizador */
        break;
    case ack_timeout:
        send_frame(ack,0,frame_expected, out_buf);  /* expira el temporizador de ack;
                                                       envía ack */
    }
    if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
}
```

Figura 3-19. Protocolo de ventana corrediza con repetición selectiva.

La recepción no secuencial introduce ciertos problemas que no se presentan en los protocolos en los que las tramas sólo se aceptan en orden. Podemos ilustrar el problema fácilmente con un ejemplo. Suponga que tenemos un número de secuencia de tres bits, por lo que se permite al emisor enviar hasta siete tramas antes de que se le exija que espere una confirmación de recepción. Inicialmente las ventanas del emisor y del receptor están como se muestra en la figura 3-20(a). El emisor ahora transmite las tramas 0 a 6. La ventana del receptor le permite aceptar cualquier trama con un número de secuencia entre 0 y 6, inclusive. Las siete tramas llegan correctamente, por lo que el receptor confirma su recepción y avanza su ventana para permitir la recepción de 7, 0, 1, 2, 3, 4 o 5, como se muestra en la figura 3-20(b). Los siete búferes se marcan como vacíos.

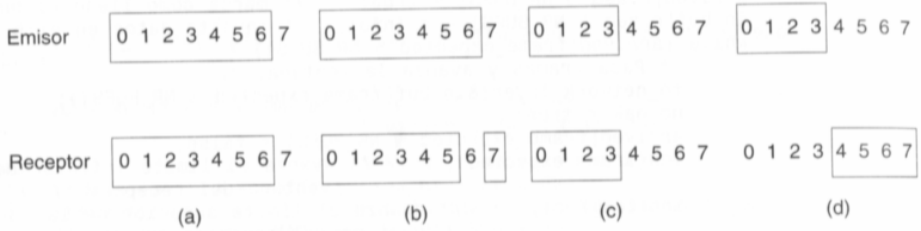


Figura 3-20. (a) Situación original con una ventana de tamaño 7. (b) Después de que se han enviado y recibido siete tramas, pero su recepción no se ha confirmado. (c) Situación inicial con un tamaño de ventana de 4. (d) Después de que se han enviado y recibido cuatro tramas, pero su recepción no se ha confirmado.

Es en este punto en el que aparece un desastre en la forma de un rayo que cae en el poste telefónico, borrando todas las confirmaciones de recepción. En algún momento termina el temporizador del emisor y retransmite la trama 0. Cuando esta trama llega al receptor, se efectúa una verificación para saber si está dentro de la ventana del receptor. Desgraciadamente, en la figura 3-20(b), la trama 0 está dentro de la nueva ventana, por lo que se acepta. El receptor envía una confirmación de recepción, superpuesta, para la trama 6, ya que se han recibido de la 0 a la 6.

El emisor se entera con beneplácito que todas sus tramas transmitidas han llegado de manera correcta, por lo que avanza su ventana y envía de inmediato las tramas 7, 0, 1, 2, 3, 4 y 5. El receptor aceptará la trama 7 y el paquete de ésta se pasará directamente a la capa de red. Inmediatamente después, la capa de enlace de datos receptora revisa si ya tiene una trama 0 válida, descubre que sí y pasa el paquete que contiene a la capa de red. En consecuencia, la capa de red obtiene un paquete incorrecto, y falla el protocolo.

La esencia del problema es que una vez que el receptor ha avanzado su ventana, el nuevo intervalo de números de secuencia válidos se traslapa con el anterior. En consecuencia, el siguiente grupo de tramas podría ser de tramas duplicadas (si se perdieron todas las confirmaciones de recepción) o de nuevas (si se recibieron todas las confirmaciones de recepción). El pobre receptor no tiene manera de distinguir entre estos dos casos.

La salida de este dilema es asegurarse que, una vez que el emisor haya avanzado su ventana, no haya traslape con la ventana original. Para asegurarse de que no haya traslape, el tamaño máximo de la ventana debe ser cuando menos de la mitad del intervalo de los números de secuencia, como en las figuras 3-20(c) y 3-20(d). Por ejemplo, si se utilizan 4 bits para los números de secuencia, éstos tendrán un intervalo de 0 a 15. Sólo ocho tramas sin confirmación de recepción deben estar pendientes en cualquier instante. De esa manera, si el receptor apenas ha aceptado las tramas 0 a 7 y ha avanzado su ventana para permitir la aceptación de las tramas 8 a 15, puede distinguir sin ambigüedades si las tramas subsiguientes son retransmisiones (0 a 7) o nuevas (8 a 15). En general, el tamaño de la ventana para el protocolo 6 será (*MAX_SEQ* + 1)/2. Por lo tanto, para números de secuencia de 3 bits, el tamaño de ventana es 4.

Una pregunta interesante es: ¿cuántos búferes deberá tener el receptor? En ninguna circunstancia puede aceptar tramas cuyos números de secuencia estén por debajo del extremo inferior o por encima del extremo superior de la ventana. En consecuencia, el número de búferes necesarios es igual al tamaño de la ventana, no al intervalo de números de secuencia. En el ejemplo anterior de un número de secuencia de 4 bits, se requieren ocho búferes, numerados del 0 al 7. Cuando llega la trama *i*, se coloca en el búfer *i* mod 8. Observe que, aun cuando *i* e (*i* + 8) mod 8 están "compitiendo" por el mismo búfer, nunca están dentro de la ventana al mismo tiempo, pues ello implicaría un tamaño de ventana de cuando menos 9.

Por la misma razón, el número de temporizadores requeridos es igual al número de búferes, no al tamaño del espacio de secuencia. Efectivamente, hay un temporizador asociado a cada búfer. Cuando termina el temporizador, el contenido del búfer se retransmite.

En el protocolo 5 se supone de manera implícita que el canal está fuertemente cargado. Cuando llega una trama, no se envía de inmediato la confirmación de recepción. En cambio, esta última se superpone en la siguiente trama de datos de salida. Si el tráfico de regreso es ligero, la confirmación de recepción se detendrá durante un periodo largo. Si hay mucho tráfico en una dirección y no hay tráfico en la otra, sólo se envían *MAX_SEQ* paquetes y luego se bloquea el protocolo, que es por lo cual dimos por hecho que siempre había tráfico de regreso.

En el protocolo 6 se corrige este problema. Tras llegar una trama de datos en secuencia, se arranca un temporizador auxiliar mediante *start_ack_timer*. Si no se ha presentado tráfico de regreso antes de que termine este temporizador, se envía una trama de confirmación de recepción independiente. Una interrupción debida al temporizador auxiliar es conoce como evento *ack_timeout*. Con este arreglo, ahora es posible el flujo de tráfico unidireccional, pues la falta de tramas de datos de regreso a las que puedan superponerse las confirmaciones de recepción ya no es un obstáculo. Sólo existe un temporizador auxiliar, y si *start_ack_timer* se invoca mientras el temporizador se está ejecutando, se restablece a un intervalo completo de temporización de la confirmación de recepción.

Es indispensable que el tiempo asociado al temporizador auxiliar sea notablemente más corto que el del temporizador usado para la terminación de tramas de datos. Esta condición es necesaria para asegurarse de que la confirmación de recepción de una trama correctamente recibida llegue antes de que el emisor termine su temporización y retransmita la trama.

El protocolo 6 utiliza una estrategia más eficiente que el 5 para manejar los errores. Cuando el receptor tiene razones para suponer que ha ocurrido un error, envía al emisor una trama de confirmación de recepción negativa (NAK). Tal trama es una solicitud de retransmisión de la trama especificada en la NAK. Hay dos casos en los que el receptor debe sospechar: cuando llega una trama dañada, o cuando llega una trama diferente de la esperada (pérdida potencial de la trama). Para evitar hacer múltiples solicitudes de retransmisión de la misma trama perdida, el receptor debe saber si ya se ha enviado una NAK para una trama dada. La variable *no_nak* del protocolo 6 es true si no se ha enviado todavía ninguna NAK para *frame_expected*. Si la NAK se altera o se pierde no hay un daño real, pues de todos modos el emisor terminará su temporizador en algún momento y retransmitirá la trama perdida. Si llega la trama equivocada después de haberse enviado y de perderse una NAK, *no_nak* será true y el temporizador auxiliar arrancará. Cuando termine, se enviará una ACK para resincronizar el estado actual del emisor con el del receptor.

En algunas situaciones, el tiempo requerido para que una trama se propague a su destino, sea procesada ahí y regrese la confirmación de recepción es (casi) constante. En estos casos, el emisor puede ajustar su temporizador para que apenas sea mayor que el intervalo de tiempo esperado entre el envío de una trama y la recepción de su confirmación. Sin embargo, si este tiempo es muy variable, el emisor se enfrenta a la decisión de establecer el intervalo en un valor pequeño (y arriesgarse a retransmisiones innecesarias) o de establecerlo en un valor grande (quedándose inactivo por un periodo largo tras un error).

Ambas opciones desperdician ancho de banda. Si el tráfico de regreso es esporádico, el tiempo antes de la confirmación de recepción será irregular, siendo más corto al haber tráfico de regreso y mayor al no haberlo. El tiempo de procesamiento variable en el receptor también puede ser un problema aquí. En general, cuando la desviación estándar del intervalo de confirmación de recepción es pequeña en comparación con el intervalo mismo, el temporizador puede hacerse "justo" y las NAKs no serán útiles. De otra manera, el temporizador deberá hacerse "holgado", y las NAKs pueden acelerar apreciablemente la retransmisión de tramas perdidas o dañadas.

Un aspecto muy relacionado con el asunto de la terminación de los temporizadores y las NAKs es cómo determinar qué trama causó una terminación del temporizador. En el protocolo 5 siempre es *ack_expected*, pues es la más antigua. En el protocolo 6 no hay ninguna manera sencilla de determinar quién ha terminado el temporizador. Suponga que ya se transmitieron las tramas 0 a 4, de modo que la lista de tramas pendientes es 01234, en orden de la más antigua a la más nueva. Ahora imagine que 0 termina su temporizador, se transmite 5 (una trama nueva), 1 termina su temporizador, 2 termina su temporizador y se transmite 6 (otra trama nueva). En este punto, la lista de tramas pendientes es 3405126, de la más antigua a la más nueva. Si todo el tráfico de entrada se pierde durante un rato (es decir, las tramas que llevan las confirmaciones de recepción), las siete tramas pendientes terminarán su temporizador en ese orden.

Para evitar que el ejemplo se complique aún más, no hemos mostrado la administración de los temporizadores. En cambio, suponemos que la variable *oldest_frame* se establece en el momento de terminación del temporizador para indicar la trama cuyo temporizador ha terminado.

ANEXO D

OBJETOS DE RED BÁSICOS EN NS-2

A continuación se describen los objetos de red básicos que forman parte de una simulación en NS-2.

Nodo

Un nodo es un objeto de la clase *node* implementada únicamente en el espacio OTcl, no en C++, ya que sólo realiza la interconexión de objetos de red sencillos como son clasificadores y agentes (implementados en C++).

- **Clasificadores:** Poseen varias salidas, pudiendo los datos de entrada optar por cualquiera de ellas; entre estos componentes están los clasificadores de direcciones y de puertos.
- **Agentes:** Representan puntos terminales en la red en donde los paquetes son creados y destruidos. Son considerados como entidades de procesamiento, por lo que son usados para implementar protocolos. Cada agente es asignado a un único puerto del nodo.

En NS-2 se pueden simular dos tipos de nodos: *unicast* y *multicast*.

- **Unicast:** Esta conformado por un clasificador de direcciones y un clasificador de puertos.
- **Multicast:** Tiene la misma estructura de un nodo *unicast* y adicionalmente un clasificador *multicast.*

En la Figura D1 se indica la estructura interna de un nodo *unicast* y *multicast*, en donde se puede observar que los nodo están compuestos por objetos de la clase *Classifier* tanto para las direcciones como para los puertos.

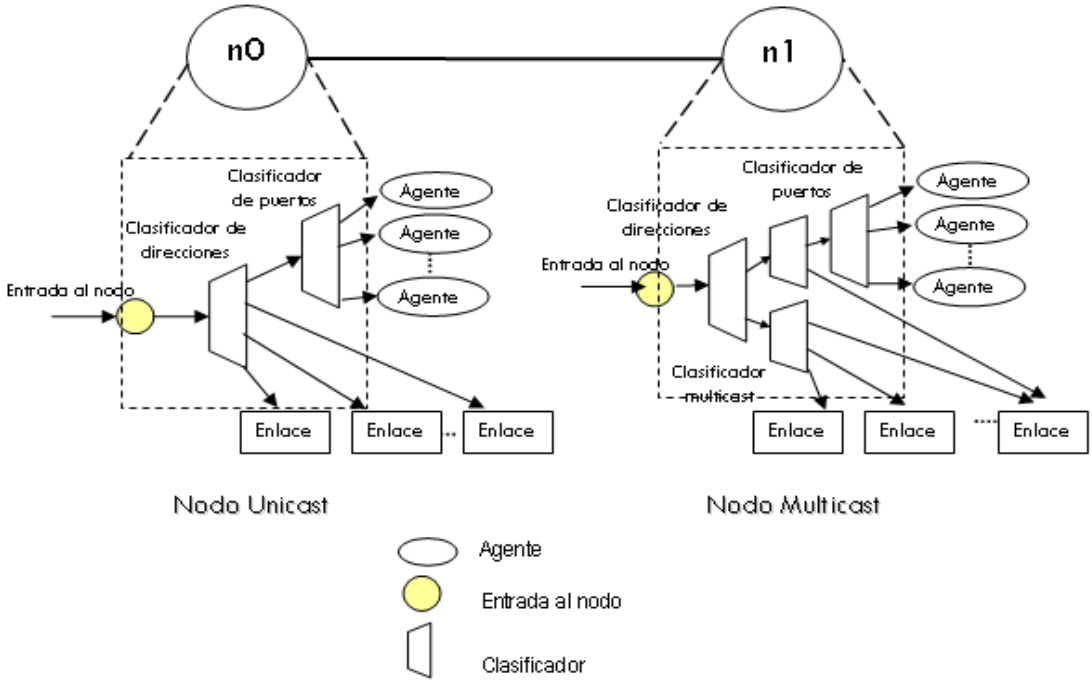


Figura D1. Estructura interna de un nodo *unicast* y de un nodo *multicast*.

Enlace

Un enlace es un objeto de la clase *link* implementada únicamente en el espacio OTcl no en C++, ya que sólo realiza la interconexión de los objetos de las clases: *Queue* (en el que serán encolados los datos), *Delay* (simula retardo en el enlace), *Agent/Null* (procesa los datos que se pierdan), *TTL* (calcula el tiempo de vida para cada paquete recibido).

Sirve para enlazar nodos y modela un canal de transmisión.

En NS-2 se pueden utilizar dos tipos de enlaces: *simplex-link* y *duplex-link.*

- **simplex-link:** Conecta dos nodos punto a punto mediante un enlace unidireccional.

La Figura D2 indica la estructura interna de un enlace *simplex-link*.

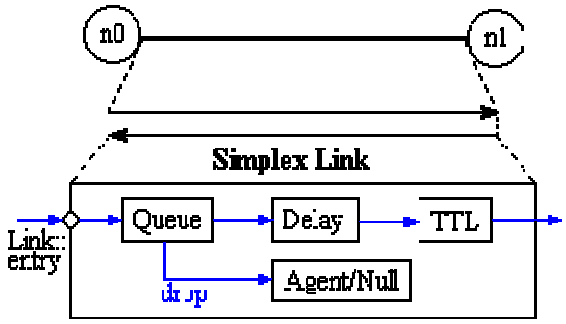


Figura D2. Estructura interna de un enlace *simplex-link*.

- **duplex-link:** Conecta dos nodos punto a punto mediante un enlace bidireccional. Éste se forma por la unión de dos enlaces *simplex-link* para conectar el nodo emisor al nodo receptor y el nodo receptor al nodo emisor.

Ruta de datos

La ruta de datos se establece en el momento en que se transmite la información a través de los objetos de red antes descritos.

En la Figura D3 se presenta la ruta de datos que se establece cuando un nodo transmite información a otro; se ha configurado un agente de nombre *Protocolo0* al nodo *n0*, y un agente *Protocolo1* al nodo *n1*; además se ha establecido un enlace *simplex-link* desde el nodo *n0* hacia el nodo *n1* y desde el nodo *n1* hacia el nodo *n0*.

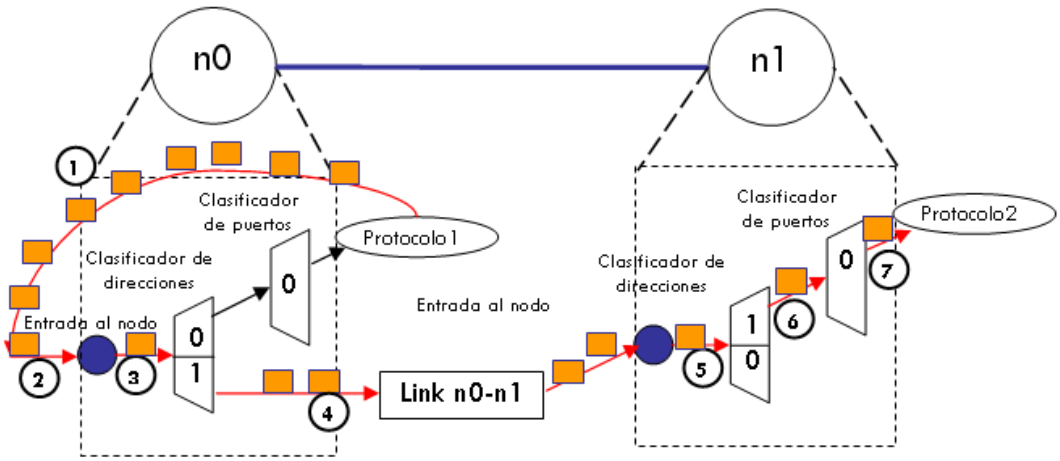


Figura D3. Flujo de datos entre dos objetos de red.