Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de un sistema de comunicaciones punto a punto con ns3

Autor: Belén Rodríguez Estévez
Tutor: Francisco Javier Payán Somet

Dep. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de un sistema de comunicaciones punto a punto con ns3

Autor:
Belén Rodríguez Estévez

Tutor:
Francisco Javier Payán Somet
Profesor titular

Dep. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo de Fin de Grado: Desarrollo de un sistema de comunicaciones punto a punto con ns3

Autor: Belén Rodríguez Estévez

Tutor: Francisco Javier Payán Somet

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis amigos

Agradecimientos

Con este proyecto acaban los que han sido cuatro años de continuo esfuerzo, superación, y, sobre todo, aprendizaje. Quiero dar las gracias a todos los que han contribuido a hacer de estos años los mejores de mi vida.

Gracias, en primer lugar, a mi tutor, Francisco Javier Payán, por haberme dado la oportunidad de realizar este trabajo y por contar con la paciencia necesaria para hacerlo.

Gracias a todos los que han sido más que compañeros, a Fran y Gabri por ayudarme continuamente y sacarme sonrisas cuando más me hacía falta. Gracias a Irene, por haber sido incondicional en este último tramo y haberme aguantado en tantas clases.

Gracias a Paula, que, desde Madrid, ha continuado dándome ánimos y distrayéndome de la constante rutina.

Gracias a Gonzalo, que ha sido, es y será mi pilar fundamental en la vida.

Y gracias, especialmente, a mi familia, que me ha apoyado desde el inicio con esta locura de ser ingeniera. A mis padres, por hacer que toda esta experiencia haya sido posible, y a mi hermano, con el que comparto tantas tonterías.

Gracias a todos.

Belén Rodríguez Estévez

Sevilla, 2017

Resumen

El objetivo de este trabajo es el diseño de un sistema de comunicaciones punto a punto mediante el uso de un software de simulación llamado ns3. Al ser un software que no se ha tratado anteriormente, es necesario adquirir una serie de conocimientos básicos sobre este.

Es por ello que, en nuestro estudio, antes de realizar simulación alguna, se llevará a cabo un proceso de aprendizaje de algunos de los múltiples aspectos de ns3. Para ello, se procederá realizando un resumen teórico del funcionamiento en cada apartado, complementándose con ejemplos realizados.

Se ha optado por ir introduciendo los ejemplos a lo largo del documento, para que la lectura sea más dinámica, y con el fin de favorecer la asimilación necesaria de los contenidos.

Abstract

The aim of this work is the design of a point to point communication system by using a simulation software called ns3. Being a software which has not been dealt with before, makes necessary to acquire some basic knowledge about it.

That is why, in our study, before doing any simulation, a learning process of some of the many aspects of ns3 will be carried out. For that purpose, we are carrying through with a theoretical summary of the operation in each section, complementing it with several examples.

It has been chosen to introduce examples throughout the document, for the reading to be more dynamic, in order to increase the understanding of the contents.

Índice

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN AL SIMULADOR NS3

Scientists discover the world that exists, engineers create the world that never was.

Los científicos descubren el mundo tal como es; los ingenieros crean el mundo que nunca ha sido.

- Theodore von Karman -

En un mundo donde las tecnologías son de uso cotidiano, resulta indispensable disponer de herramientas que nos permitan estudiar su comportamiento. Para ello podemos contar con tres métodos de estudio: analítico, experimental y de simulación.

Si empleamos técnicas matemáticas que describan el funcionamiento del sistema, será un método analítico. La experimentación la utilizaremos cuando dispongamos de un sistema real, realizando pruebas sucesivas sobre él. Cuando no tengamos a nuestra disposición dicho sistema y los modelos matemáticos sean difíciles de abordar, podemos recurrir a la simulación. Esto consiste en realizar una representación de un sistema real y llevar a cabo experimentos con dicho modelo, con el fin de evaluar su comportamiento.

1.1 Simulador ns3

El ns3 es un simulador de redes basado en eventos discretos. Se emplea, principalmente, en ámbitos educativos y de investigación. La infraestructura software ns3 fomenta el desarrollo de modelos de simulaciones que son lo suficientemente realistas como para ser usados como un emulador de redes a tiempo real.

Cada tres meses se lanza una nueva versión de ns3 con nuevos modelos desarrollados por investigadores y comprobaciones realizadas por terceras personas, con el fin de ofrecer la máxima calidad posible.

ns3 es una librería C++ que proporciona modelos de simulación de redes implementados como objetos C++ y con interfaces en phyton. Los usuarios interaccionan con esta librería mediante aplicaciones C++ o phyton que instancian unos modelos de simulación para montar el escenario de interés, entrar en el bucle principal de la simulación, y salir cuando la simulación esté completa.

Este software es un programa de código abierto y depende de las continuas contribuciones de la comunidad de usuarios e investigadores. Es posible colaborar con este proyecto mediante varias vías:

- Listas de discusión.
- Reportar errores que se crea que se han encontrado.
- Proporcionar tutoriales o contenido a la wiki.
- Aportar código, tanto si se ha desarrollado un nuevo modelo como modificado uno existente.
- Mantener el simulador.

La página web principal está localizada en http://www.nsnam.org/ y proporciona acceso a información básica sobre el sistema ns3. La documentación detallada está disponible en http://www.nsnam.org/documentation/ donde, principalmente, viene una lista de todas las clases con sus respectivos atributos y métodos. Hay una wiki que complementa a la web principal de ns3, que está en http://www.nsnam.org/wiki/ donde se encuentran respuestas a las preguntas frecuentes, así como guías de solución de problemas, códigos de terceras personas… El código fuente está disponible en https://code.nsnam.org/. Doxygen es la herramienta estándar para generar

documentación de tipo C++. El software ns3 dispone también de dicho instrumento en el enlace https://www.nsnam.org/doxygen/index.html.

1.2 Instalación

Se instala en cualquier plataforma Linux como, por ejemplo, Ubuntu. Lo descargamos de la página web oficial y descomprimimos la carpeta, o realizamos lo mismo mediante comandos:

```
wget http://www.nsnam.org/release/ns-allinone-3.26.tar.bz2
tar xjf ns-allinone-3.26.tar.bz2
```

En este caso, la versión más reciente del software es la 3.26. Para versiones posteriores, basta con cambiar el número por la versión actualizada que queramos descargarnos.

El código desarrollado tiene que ser introducido en una carpeta del directorio ./scratch. Dentro de dicha carpeta, solo puede haber una función *main*, de modo que el resto de ficheros debe depender de aquel que contenga dicha función. Para compilar, enlazar y ejecutar un archivo, tenemos que situarnos en el directorio padre de ./scratch, es decir, ./ns-allinone-3.26. Hay varios mecanismos para realizarlo, aunque el que nosotros vamos a utilizar es el *waf*. El compilador más conocido es *make*, sin embargo, también es uno de los más difíciles de emplear. Por esta causa, se ha desarrollado una nueva generación de compiladores, basados en phyton, tales como *waf*, en el que no se necesita conocer dicho lenguaje para utilizar el compilador.

Para ello, hay que instalar unos prerrequisitos necesarios para los lenguajes que vamos a emplear que, en Ubuntu, se puede hacer de la siguiente forma:

```
apt-get install gcc g++ python
apt-get install gcc g++ python python-dev
apt-get install mercurial python-setuptools git
apt-get install qt4-dev-tools libqt4-dev
apt-get install cmake libc6-dev libc6-dev-i386 g++-multilib
apt-get install gdb valgrind
apt-get install gsl-bin libgsl2 libgsl-dev
apt-get install flex bison libfl-dev
apt-get install tcpdump
apt-get install sqlite sqlite3 libsqlite3-dev
apt-get install libxml2 libxml2-dev
apt-get install libgtk2.0-0 libgtk2.0-dev
apt-get install vtun lxc
apt-get install uncrustify
apt-get install doxygen graphviz imagemagick
apt-get install texlive texlive-extra-utils texlive-latex-extra
texlive-font-utils texlive-lang-portuguese dvipng
apt-get install python-sphinx dia
apt-get install python-pygraphviz python-kiwi python-pygoocanvas
libgoocanvas-dev ipython
apt-get install libboost-signals-dev libboost-filesystem-dev
apt-get install openmpi-bin openmpi-common openmpi-doc libopenmpi-dev
```

Aunque no todos son fundamentales para el funcionamiento, sí son recomendables. Para realizar la configuración del compilador, encontramos dos modos alternativos: optimizado y depurador. En el primer caso, los comandos son los siguientes:

```
./waf clean
./waf configure --build-profile=optimized --enable-examples --enable-
tests
```

Sin embargo, es recomendable usar el modo depurador para trabajar, pues así se activan los avisos que puede proporcionar el sistema, que veremos en el apartado 3-4:

```
./waf clean
./waf configure --build-profile=debug --enable-examples --enable-tests
```

Una vez instalado todo correctamente, ya podemos compilar un programa. Con este fin, creamos una carpeta en el directorio ./scratch, a la que denominamos, por ejemplo, *proyecto,* y situamos el código fuente en ella. El mecanismo de compilación y ejecución del programa se realiza introduciendo el siguiente comando desde el terminal

```
./waf --run proyecto
```

siendo *proyecto* el nombre de la carpeta que hemos creado en el directorio ./scratch. Hay que tener en cuenta que, aunque solo ejecute los archivos que le hayamos especificado, compila todas las carpetas que existan en dicho directorio. Esto implica que, a pesar de que los archivos de la carpeta *proyecto* sean correctos, si tenemos otra carpeta *proyecto2* con algún archivo erróneo, al ejecutar la orden anterior, nos saltará un error.

2 LENGUAJE C++

La chance c'est quand la préparation et l'opportunité se rencontré.

Suerte es lo que sucede cuando la preparación y la oportunidad se encuentran y fusionan.

- Voltaire -

C++ es un lenguaje de programación orientado a objetos (POO), en el que los programas se construyen a partir de componentes individuales llamados objetos. Un objeto encapsula los datos y las operaciones que se pueden llevar a cabo sobre esos datos. En general, la programación se resuelve comunicando dichos objetos a través de mensajes. Su principal ventaja es la reutilización de códigos. La extensión de los archivos empleados con código fuente es .cc.

2.1 Clase

Define la generalización de los objetos, incluyendo sus atributos (características, campos) y sus métodos (las cosas que puede hacer, propiedades). Una clase describe el comportamiento de una familia de objetos.

Los datos solo pueden ser modificados utilizando los métodos del objeto. El conjunto de métodos de un objeto que otros objetos pueden llamar se conoce como la interfaz.

Los atributos y los métodos pueden ser de dos clases: públicos y privados.

Privados (private): son los elementos que solo pueden ser utilizados por miembros de esa clase. Habitualmente todos los atributos suelen ser privados.

Públicos (public): son miembros que pueden ser utilizados por objetos de cualquier clase. Determinan la interfaz de la clase.

2.2 Objetos

Representan un elemento individual e identificable, con un comportamiento bien definido en el dominio del problema. Un objeto de una determinada clase se crea cuando se define una variable de dicha clase.

El estado de un objeto viene caracterizado por sus propiedades o atributos. Estos deben tener un valor, que puede variar a lo largo de la vida de un objeto. Cada objeto tiene sus propios datos.

Los métodos son las habilidades de un objeto, es decir, las funciones a las que puede llamar una clase. Un método afecta a un solo objeto en particular.

Ejemplo: La clase Perro podría considerar como propiedades la raza y color; y como métodos, la habilidad de ladrar y sentarse. El perro llamado Rayo es un objeto de la clase Perro, distinto del perro llamado Trueno, pero ambos comparten las características de dicha clase, con distintos valores cada uno.

2.3 Constructor

Un constructor es un método especial de una clase, que es llamado automáticamente siempre que se crea un objeto de dicha clase. Su función es iniciar el objeto. Debe tener el mismo nombre que la clase.

2.4 Polimorfismo

Es la habilidad de los objetos de responder con distinto comportamiento ante un mismo mensaje. Se pueden definir varios métodos (o constructores) con el mismo nombre que difieran en su definición. A esto se le llama sobrecarga. El compilador distingue cuál debe utilizar en cada caso mediante el número o tipo de parámetros asociados. Los métodos sobrecargados pueden devolver valores de tipos distintos, pero no se pueden diferenciar únicamente en ello.

2.5 Herencia

Incrementa la reutilización de código, ya que permite que una clase pueda volver a emplear parte de la conducta de otra clase. Para ello, tenemos una jerarquía, donde las clases pueden heredar atributos y métodos de las clases padre o superclases y añadir las suyas propias. Por tanto, las superclases son generalizaciones de las subclases o hijos que, a su vez, son clases especializadas de éstas. La herencia puede ser simple, si una clase hereda exclusivamente de otra clase; o múltiple, si una clase tiene varias superclases.

Una clase hija hereda todos los miembros de la superclase, excepto los constructores. Sin embargo, la subclase solo tiene acceso directo a los miembros *public* de su padre, pese a que también hereda los *private*.

Una subclase puede añadir sus propios métodos y atributos. Si el nombre de alguno de ellos coincide con el de un miembro heredado, este último quedará oculto, ya que se sobreescribe.

Los miembros heredados por una subclase pueden ser heredados por una subclase de esta, lo que se llama propagación de herencia.

2.6 This

Cada objeto mantiene una copia propia de los atributos, pero no de los métodos de la clase, de los cuales solo existe una copia para todos los objetos de dicha clase. Es decir, todos los objetos de una misma clase comparten el mismo método. Para que un método pueda hacer mención al objeto que lo invocó se utiliza *this*, que es un puntero implícito. Es una referencia al objeto sobre el que ha sido llamado el método, y puede ser utilizada dentro de cualquier método.

2.7 Estructuras de control

Como cualquier otro lenguaje, C++ nos proporciona una serie de estructuras de control para poder desarrollar los programas:

- Selección: también llamada estructura condicional, permite que ciertas sentencias se ejecuten o no en función de una determinada condición.

```
if (condicion)
{sentencia1}
else
{sentencia2}
```

```
switch (condicion)
{
  case valor1:
     sentencia1;
     break;
  case valor2:
     sentencia2;
     break;
...
  default:
     sentencia;
}
```

```
try
  {sentencias}
catch (condicion1)
  {sentencia1}
catch (condicion2)
  {sentencia2}
...
```

Las estructuras de tipo try-catch se emplean para capturar excepciones, que, lanzadas por la palabra reservada *throw*, se encuentren en las sentencias ubicadas en *try*.

- Iteración: se repite un conjunto de instrucciones en función de una condición.

`while (condicion) {sentencias}`	`do {sentencias} while (condicion)`	`for (inicio; condicion; iteracion) {sentencias}`

- Salto: no se recomienda su uso, salvo en ocasiones excepcionales.

`break`	`return`	`continue`	`goto`

2.8 Tipos de variables

Los tipos de variables definidos en C++ son los mostrados en la Tabla 2-1.

Tabla 2-1. Tipos de variables, espacio y rango de C++

Tipo	Espacio	Rango
int	32 bits	-2.147.483.648 a 2.147.483.647
short int	16 bits	-32.768 a 32.767
unsigned int	32 bits	0 a 4.294.967.295
char	8 bits	-128 a 127
float	32 bits	$3'4x10^{-38}$ a $3'4x10^{38}$
double	64 bits	$1'7x10^{-308}$ a $1'7x10^{308}$
long double	80 bits	$3'4x10^{-4932}$ a $3'4x10^{4932}$
bool	8 bits	0 a 1

En la librería de cabecera <*cstdin*>, tenemos otra serie definida de tipos de variables, de las que destacaremos las siguientes, muy usadas posteriormente en el desarrollo de programas en ns3:

Tabla 2-2. Tipos de variables, espacio y rango de cstdin

Tipo	Espacio	Rango
uint8_t	8 bits	0 a 255
uint16_t	16 bits	0 a 65.535
uint32_t	32 bits	0 a 4.294.967.295
uint64_t	64 bits	0 a 18.446.744.073.709.551.615

Dichas variables tienen sus respectivas equivalencias con signo, siendo int8_t, int16_t, int32_t e int64_t.

2.9 Interacción con pantalla

Debido a que este lenguaje es una evolución de C, mantiene las técnicas de este último a la hora de imprimir y escanear por pantalla. Es decir, utiliza las funciones printf y scanf. Para imprimir por pantalla utilizamos la primera de la siguiente forma:

```
printf("texto, %d\n", variable);
```

donde la secuencia de escape '\n' representa el carácter 'nueva línea' y es opcional. Del mismo modo, se puede emplear '\t', que representa el carácter tabulador horizontal.

Por otra parte, '%d' es un carácter de conversión, que sirve para imprimir por pantalla el valor de la *variable* que especifiquemos posteriormente. Destacaremos que el carácter *d* se empleará para variables de tipo entero, *f* para double y *s* para cadenas de caracteres. De forma natural, si no queremos incluir el valor de alguna variable, podemos obviar dicho término.

El uso de scanf se hace del mismo modo que hemos explicado para printf. El principal inconveniente del uso de estas funciones es que se debe especificar el tipo de variable que queremos imprimir o escanear por pantalla.

En la biblioteca estándar de C++, *std,* encontramos definidas otras funciones para la interacción con el usuario. Para el uso de estas no necesitamos conocer el tipo de variable a imprimir o escanear, lo que representa una gran ventaja frente a las anteriores. Es por este motivo que, a lo largo del documento, vamos a utilizar estas en el caso de necesitarlas.

Tabla 2-3. Funciones de interacción por pantalla de la biblioteca estándar de C++

std::cout	Permite imprimir por pantalla cadenas de texto y variables, concatenando unas con otras con los caracteres '<<'.
std::endl	Introduce un carácter 'nueva línea' al final de la sentencia impresa por *cout*.
std::cin	Obtiene un valor introducido por el usuario.

Ejemplo: Tenemos una variable llamada Edad, cuyo contenido es 27. Para imprimir dicho valor por pantalla, ejecutaremos una de las siguientes sentencias, siendo preferible la segunda:

printf("La edad del alumno es %d\n", edad);

std::cout << "La edad del alumno es " << edad << std::endl;

Cabe destacar que los tipos de variables definidos en la cabecera *cstdin* no se pueden imprimir por pantalla con esta opción, sino que debe hacerse una conversión a uno de los tipos por defecto del lenguaje, normalmente int.

3 PRIMEROS PASOS

An expert is a man who has made all the mistakes, which can be made, in a very narrow field.

Un experto es una persona que ha cometido todos los errores posibles en un determinado campo.

- Niels Bohr -

En este capítulo vamos a tratar los elementos y clases fundamentales del software de simulación empleado.

3.1 Namespace ns3

Los *namespaces* definen los contextos en los que se desarrolla el código. Si no especificamos ninguno, estaremos en el espacio global. Los proyectos de este software se desarrollan en un *namespace* denominado ns3. Esto agrupa todas las declaraciones relacionadas con ns3 en un contexto distinto al espacio global. El uso de este espacio ahorra la necesidad de introducir 'ns3::' cada vez que se haga alusión a un elemento definido por el software de interés.

3.2 Modules

Los distintos archivos de cabecera que definen las clases de ns3 se pueden agrupar en módulos para facilitar su utilización. Esto se debe a que una gran mayoría de clases incorporan dependencias de otras, por lo que sería necesario incluir todos los archivos de cabecera. Los módulos que existen en este software se definen en la tabla siguiente.

Tabla 3-1. Listado de módulos de ns3

6LoWPAN	AODV Routing	Applications
BRITE Topology Generator	Bridge Network Device	CSMA Layout Helpers
CSMA Network Device	Click Routing	Configuration Store/Load Constants
Core	DSDV Routing	DSR Routing
Energy Models	File Descriptor Network Device	Flow Monitor
Internet	Internet Applications	LR-WPAN models
LTE Models	MPI Distributed Simulation	Mesh Device
Mobility	Network	Network Animation

Tabla 3-1. Listado de módulos de ns3 (continuación)

Nix-Vector Routing	OLSR Routing	OpenFlow Switch Device
Point-To-Point Network Device	Point-to-Point Layout Helpers	Propagation Models
Spectrum Models	Statistics	Tap Bridge Network Device
Topology Input Readers	Traffic-control	UAN Models
Utils	Virtual Device	Visualizer
WAVE Module	WiMAX Models	Wifi Models

Para incluir dichos módulos en el código que desarrollemos, se realizará del mismo modo que la introducción de un fichero de cabecera habitual:

```
#include <ns3/nombre-module.h>
```

A lo largo del trabajo emplearemos los módulos *point-to-point-module* (que se refiere al módulo *Point-To-Point Network Device*) cuando utilicemos dispositivos punto a punto, así como *network-module* (el cual incluye clases como *Packet* y *Sequencenumber* que veremos posteriormente). Además de los dos mencionados, el módulo principal de ns3 es *core-module*, que incluiremos en todos los ficheros que desarrollemos. Esto se debe a que abarca las clases principales necesarias para realizar un programa en este software, tales como la clase *Time* y la clase *Simulator* entre otras.

3.3 Simulator

La clase principal del ns3 es *Simulator*, la cual controla la programación de los eventos en la simulación. Esta clase está definida en el fichero de cabecera *simulator.h*. Algunas de sus funciones más destacadas son:

Simulator::Run(): inicia la ejecución de la simulación. La simulación no parará hasta que se dé uno de los siguientes casos:

- No hay más eventos programados.
- Se llama a *Simulator::Stop*() dentro del procesamiento de un evento.
- Expira el tiempo de *Simulator::Stop*(Time *retardo).*

Simulator::Stop(): detiene la simulación en el instante en que se ejecuta la sentencia.

Simulator::Stop(Time *retardo***)**: detiene la simulación cuando ha transcurrido un tiempo *retardo.*

En simulaciones recurrentes, donde la ejecución de un evento conlleva la llamada a otro o una autoprogramación del mismo evento, es indispensable detener la simulación con alguna de las dos funciones anteriores. Es importante realizar la llamada a alguno de ellos antes de empezar el programa con *Simulator::Run*, ya que, en caso contrario, el programa podría ejecutarse sin devolver el control al programa principal y, por tanto, no alcanzar nunca el *Stop*.

Simulator::Now(): indica el instante virtual en el que se encuentra en la simulación (en medida de tiempo). Por defecto, lo muestra en microsegundos. Para presentarlo en otra medida, por ejemplo, en segundos, bastaría con *Simulator::Now().GetSeconds()*. Para más detalles ver apartado 3.9.

Simulator::Cancel(EventId *identificador***)**: cancela el evento con id *identificador*, es decir, cuando expire el tiempo establecido anteriormente con *Schedule*, no se ejecutará el evento.

Simulator::Remove(EventId *identificador***)**: tiene el mismo efecto que la función anterior, aunque con un algoritmo más elaborado. Cabe destacar que no se pueden eliminar eventos asociados al tiempo de *destroy*.

Simulator::Destroy(): libera memoria.

Simulator::GetContext(): devuelve el contexto en el que se encuentra la simulación en dicho instante. En ns3 se entiende por contexto los diferentes procesos lógicos.

Simulator::GetDelayLeft(): indica el tiempo que le resta a un evento para ejecutarse.

Simulator::GetMaximumSimulationTime(): obtiene el tiempo máximo representable en la simulación, es decir, el retardo máximo que puede asociarse a la ejecución de un evento.

Simulator::GetSystemId(): devuelve el identificador asociado con la simulación correspondiente.

Simulator::IsExpired(EventId *identificador***)**: nos indica si el evento ya ha sido ejecutado o cancelado. Se dice que un evento ha expirado cuando el *Schedule* asociado a él ya se ha efectuado.

Simulator::IsFinished(): verifica si la simulación ha finalizado.

3.3.1 Programación de eventos

La programación de eventos es elemento clave en todo este proceso, puesto que, a través de ella, podemos disponer que cada acción se efectúe en un tiempo concreto. Esto nos permite obtener un conjunto de acciones diferenciadas temporalmente y poder gestionarlas en función de nuestros intereses.

Simulator::Schedule(Time *retardo***, Ptr<EventImpl>** *evento***)**: programa un evento. Cuando ha pasado un tiempo *retardo*, ejecuta el evento *evento* que, normalmente es una función.[1]

Simulator::ScheduleDestroy(Ptr<EventImpl> *evento***)**: plafinica la realización de un evento en el tiempo en que la función *Destroy* es llamada. Todos los eventos asociados a dicho tiempo siguen un modelo FIFO (*First In First Out*), en el que, el primero que se programó, es el primero en ejecutarse.

Simulator::ScheduleNow(Ptr<EventImpl> *evento***)**: sitúa en el tiempo actual la llamada a dicho evento. Al igual que el anterior, el orden es FIFO, una vez se han ejecutado los eventos normales del programa.

Simulator::ScheduleWithContext(uint32_t *contexto***, Time** *retardo***, Ptr<EventImpl>** *evento***)**: se emplea cuando se quiere programar un evento que no está situado en el mismo contexto (típicamente otro archivo). Si no se sabe el contexto en el que se halla dicho evento, el valor 0xffffffff significa que el contexto no está especificado.

Habitualmente, lo que programamos son eventos, pero también se pueden programar funciones u objetos de la misma forma. Si el evento asociado o similar necesita que se le pasen parámetros, se añadirán como el tercer campo y sucesivos de *Simulator::Schedule*. Admite hasta seis argumentos más.

3.3.2 Ejemplos

A continuación, mostramos unos ejemplos sencillos donde ilustramos estos conceptos.

En el primer ejemplo, mostrado en la Figura 3-1, hemos programado dos eventos, uno a los 10 segundos y otro a los 20. Cada uno de ellos es una función que imprime por pantalla sendos mensajes. Como no hay ninguna llamada a *Simulator::Stop*, la simulación se detendrá cuando no haya más eventos programados.

Por otro lado, en el segundo código, hemos introducido la función *Simulator::Stop* para que detenga la simulación a los 15 segundos. Por tanto, el segundo mensaje no se debe imprimir, pues se cancela la simulación antes de que pase el tiempo establecido para la misma.

[1] Para conocer el funcionamiento de *EventImpl,* consultar el apartado 3.11.2.

```cpp
#include <ns3/core-module.h>

using namespace ns3;

void hola();
void adios();

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

    Simulator::Schedule(Seconds(10), &hola);
    Simulator::Schedule(Seconds(20), &adios);
    Simulator::Run();
    Simulator::Destroy();

return 0;
}

void hola ()
{
    printf("Hola mundo en %f s\n", Simulator::Now().GetSeconds());
}

void adios()
{
    std::cout << "Adios mundo en " << Simulator::Now().GetSeconds()
              << "s" << std::endl;
```

Figura 3-1: Código de ejemplo1.cc

```cpp
#include <ns3/core-module.h>

using namespace ns3;

void adios();
void hola();

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

    Simulator::Schedule(Seconds(10), &hola);
    Simulator::Schedule(Seconds(20), &adios);
    Simulator::Stop(Seconds(15));
    Simulator::Run();
    Simulator::Destroy();

return 0;
}

void hola ()
{
    std::cout << "Hola mundo en " << Simulator::Now().GetSeconds()
              << "s" << std::endl;
}

void adios()
{
    std::cout << "Adios mundo en " << Simulator::Now().GetSeconds()
              << "s" << std::endl;
}
```

Figura 3-2: Código de ejemplo1-2.cc

Podemos obtener el tiempo en distintas unidades, desde los años hasta los femtosegundos. Para evitar tener que transformar el tiempo siempre a las mismas unidades, podemos ajustar la resolución del reloj a la unidad que queramos, siendo la primera sentencia que deberíamos poner en la función principal. La orden sería de la siguiente forma:

```
Time::SetResolution(Time::S);
```

En este caso, ajustamos la resolución a los segundos (*S*), y el tiempo se mostrará por defecto en esas unidades.

La última forma de que finalice la simulación es obligarla a detenerse durante el procesamiento de un evento, es decir, llamar a *Simulator::Stop* dentro de una función que hemos establecido como evento. Esto se ilustra en la siguiente figura.

```
#include <ns3/core-module.h>

using namespace ns3;

void hola();
void adios();

int main(int argc, char*argv[])
{
    Time::SetResolution(Time::S);

    Simulator::Schedule(Seconds(10), &hola);
    Simulator::Schedule(Seconds(20), &adios);
    Simulator::Run();
    Simulator::Destroy();

return 0;
}

void hola ()
{
    std::cout << "Hola mundo en " << Simulator::Now()
              << std::endl;
    Simulator::Stop();
}

void adios()
{
    std::cout << "Adios mundo en " << Simulator::Now()
              << std::endl;
}
```

Figura 3-3: Código de ejemplo1-3.cc

Otra modificación que hemos realizado es cancelar un evento. En este caso, al cancelar el evento tras haberlo creado, parece que no tiene ninguna utilidad, pero, con vistas a un programa más extenso, puede cancelarse un evento; por ejemplo, tras no cumplirse la condición de un bucle *if*.

```
#include <ns3/core-module.h>

using namespace ns3;

void hola();
void adios();
void cancelar();

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

    Simulator::Schedule(Seconds(10), &hola);
    Simulator::Schedule(Seconds(20), &adios);
    EventId id = Simulator::Schedule(Seconds(15), &cancelar);
    Simulator::Cancel(id);
    Simulator::Run();
    Simulator::Destroy();

return 0;
}

void hola ()
{
    std::cout << "Hola mundo en " << Simulator::Now().GetSeconds()
              << "s" << std::endl;
}

void adios()
{
    std::cout << "Adios mundo en " << Simulator::Now().GetSeconds()
              << "s" << std::endl;
}

void cancelar()
{
   std::cout << "No se debe ejecutar" << std::endl;

}
```

Figura 3-4: Código de ejemplo1-4.cc

3.4 Registros

El software proporciona una serie de macros que permite a los desarrolladores mostrar mensajes por línea de comandos. Hay varios niveles:

LOG_ERROR: mensaje de error. Macro asociada NS_LOG_ERROR.

LOG_WARN: mensaje de aviso. Macro asociada NS_LOG_WARN.

LOG_DEBUG: mensaje de depuración. Macro asociada NS_LOG_DEBUG.

LOG_INFO: mensaje sobre el progreso del programa. Macro asociada NS_LOG_INFO.

LOG_FUNCTION: mensaje descriptivo de función. Macros asociadas: NS_LOG_FUNCTION (para cualquier función) y NS_LOG_NOARGS (para funciones de tipo static).

LOG_LOGIC: mensaje sobre la lógica del programa. Macro asociada NS_LOG_LOGIC.

LOG_ALL: todo lo anterior.

Si añadimos _LEVEL tras LOG, estaremos incluyendo desde ese nivel hacia atrás. Por ejemplo, NS_LOG_LEVEL_DEBUG, incluirá los mensajes de error, aviso y depuración. Lógicamente, LOG_LEVEL_ALL y LOG_ALL son equivalentes.

Todas estas macros van asociadas a un componente mediante **NS_LOG_COMPONENT_DEFINE(char*** *nombre***)**. Esta sentencia se sitúa después de haber creado el namespace ns3. Lo normal es que, por cada fichero, definamos un componente distinto.

Esto queda ilustrado en el siguiente ejemplo.

```
#include <ns3/core-module.h>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("ejemplo2");

void hola();
void adios();
void cancelar();

int main(int argc, char*argv[])
{
    Time::SetResolution(Time::S);

    Simulator::Schedule(Seconds(10), &hola);
    Simulator::Schedule(Seconds(20), &adios);
    EventId id = Simulator::Schedule(Seconds(15), &cancelar);
    Simulator::Cancel(id);
    Simulator::Run();
    Simulator::Destroy();

return 0;
}

void hola ()
{
    NS_LOG_DEBUG("Ha entrado en la funcion 'hola'");
}

void adios()
{
    NS_LOG_INFO("Adios mundo en " << Simulator::Now());
}

void cancelar()
{
    NS_LOG_ERROR("No puede entrar en esta funcion");
}
```

Figura 3-5: Código de ejemplo2.cc

Como podemos ver, hemos definido el componente '*ejemplo2*' y una serie de mensajes de depuración, información y error. Para controlar los mensajes que se muestran en pantalla, se usa la terminal. Antes de compilar y ejecutar el fichero mediante el ./waf, se especifican los mensajes que se quieran mostrar de la siguiente manera:

```
export NS_LOG=”nombrecomponente=nivel”
```

Si se quieren mostrar todos los registros de un componente, basta con solo nombrarlo, mientras que, si se quieren concatenar distintos niveles o varios componentes, se usa el signo |, de la siguiente forma:

```
export NS_LOG=”nombrecomponente_1=nivel | nombrecomponente_2=nivel”
export NS_LOG=”nombrecomponente=nivel_1|nivel_2”
```

Para mostrar como prefijos el tiempo o la función en la que se llaman las macros, añadimos del mismo modo los elementos *prefix_func* y *prefix_time*

```
export NS_LOG=”nombrecomponente=nivel|prefix_func|prefix_time”
```

Se pueden desactivar los registros si no especificamos nada, es decir, con las “” vacías o sin ellas

```
export NS_LOG=””                    export NS_LOG=
```

```
salas@localhost:~/ns-allinone-3.26/ns-3.26$ export NS_LOG="ejemplo2"
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run prueba
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.252s)
10s -1 ejemplo2:hola(): [DEBUG] Ha entrado en la funcion 'hola'
20s -1 ejemplo2:adios(): [INFO ] Adios mundo en +20.0s
salas@localhost:~/ns-allinone-3.26/ns-3.26$ export NS_LOG="ejemplo2=debug"
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run prueba
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (4.152s)
Ha entrado en la funcion 'hola'
salas@localhost:~/ns-allinone-3.26/ns-3.26$ export NS_LOG="ejemplo2=info"
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run prueba
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.212s)
Adios mundo en +20.0s
salas@localhost:~/ns-allinone-3.26/ns-3.26$ export NS_LOG="ejemplo2=debug|info"
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run prueba
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.229s)
Ha entrado en la funcion 'hola'
Adios mundo en +20.0s
salas@localhost:~/ns-allinone-3.26/ns-3.26$ export NS_LOG=""
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run prueba
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.203s)
salas@localhost:~/ns-allinone-3.26/ns-3.26$
```

Figura 3-6: Terminal para ejemplo2.cc

El *namespace* ns3 nos proporciona una serie de funciones para trabajar con estas macros.

LogComponentEnable(char * *nombre***, LogLevel** *nivel***)**: permite activar las macros de dicho componente al nivel especificado.

LogComponentEnableAll(LogLevel *nivel***)**: habilita los niveles indicados para todos los componentes.

LogComponentDisable(char * *nombre***, LogLevel** *nivel***)**: desactiva las macros del nivel especificado.

LogComponentDisableAll(LogLevel *nivel***)**: deshabilita dichas macros de todos los componentes.

De esta manera, podemos controlar lo que mostramos por pantalla ejecutando el programa de la manera habitual, sin necesidad de incluir el *export* cada vez.

```
#include <ns3/core-module.h>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("ejemplo2");

void hola();
void adios();
void cancelar();

int main(int argc, char*argv[])
{
    Time::SetResolution(Time::S);

    LogComponentEnable("ejemplo2", LOG_LEVEL_INFO);
    Simulator::Schedule(Seconds(10), &hola);
    Simulator::Schedule(Seconds(20), &adios);
    EventId id = Simulator::Schedule(Seconds(15), &cancelar);
    Simulator::Cancel(id);
    Simulator::Run();
    Simulator::Destroy();

return 0;
}

void hola ()
{
    NS_LOG_DEBUG("Ha entrado en la funcion 'hola'");
}

void adios()
{
    LogComponentDisableAll(LOG_ALL);
    NS_LOG_INFO("Adios mundo en " << Simulator::Now());
}

void cancelar()
{
    LogComponentEnable("ejemplo2", LOG_ERROR);
    NS_LOG_ERROR("No puede entrar en esta funcion");
}
```

Figura 3-7: Código de ejemplo2-2.cc

Si ejecutamos el ejemplo anterior, podemos comprobar que, aunque deberían imprimirse todos los errores de información, depuración, aviso y error, hemos desactivado las macros en la función *adios*.

3.5 Línea de comandos

Al igual que en otros lenguajes de programación, podemos programar la simulación para que admita parámetros por línea de comandos. Para ello, haremos uso de la clase *CommandLine* de ns3. Vamos a destacar tres de sus atributos:

CommandLine.Usage(String *descripcion***)**: describe el uso de dicho programa.

CommandLine.AddValue(String *nombrevariable***, String** *descripcion***,** *nombrevariable***)**: define las variables a introducir por línea de comandos. El segundo campo es una breve descripción sobre la variable a la que se le puede alterar el valor. El primer y tercer campo deben ser iguales (aunque el primero debe ser una cadena de caracteres y el tercero una variable), ya que el primero se utiliza para el comando *--help,* que explicaremos posteriormente, y en el otro se almacena el valor introducido. En este último, también se puede realizar una

llamada a otra función mediante *Callback*, donde se almacenará el valor.

CommandLine.Parse(argc, argv): analiza los elementos del programa.

```
#include <ns3/core-module.h>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("ejemplo3");


int main(int argc, char*argv[])
{
    int dia=1;
    char mes[12] = "marzo";


    CommandLine cmd;

    cmd.Usage("ejemplo3.cc :
              Como introducir parametros por la linea de comandos.");
    cmd.AddValue("dia", "Dia en el que estamos", dia);
    cmd.AddValue("mes", "Mes en el que estamos", mes);
    cmd.Parse(argc, argv);

    std::cout << "Hoy es día "<< dia << " de " << mes << std::endl;

    Simulator::Run();
    Simulator::Destroy();

return 0;
}
```

Figura 3-8: Código de ejemplo3.cc

Hay que recordar darle valores por defecto a las variables, para que no dé error si no se introducen valores por línea de comandos. Para poder introducir parámetros por la terminal, debemos saber qué parámetros se pueden variar y los tipos de variables que son cada uno. Para ello, contamos con una ayuda, que nos imprimirá en pantalla las descripciones del programa y las diversas variables que hayamos especificado, así como el nombre de cada una de ellas, con el fin de poder modificar su valor. Basta con introducir en la terminal el siguiente comando:

```
./waf –run=”carpetadelproyecto --help”
```

Lo podemos ver ilustrado en el ejemplo3.cc.

```
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run="prueba --help"
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (2.858s)
prueba [Program Arguments] [General Arguments]

ejemplo3.cc : Como introducir parametros por la linea de comandos.

Program Arguments:
    --dia:  Día en el que estamos [1]
    --mes:  Mes en el que estamos [marzo]

General Arguments:
    --PrintGlobals:              Print the list of globals.
    --PrintGroups:               Print the list of groups.
    --PrintGroup=[group]:        Print all TypeIds of group.
    --PrintTypeIds:              Print all TypeIds.
    --PrintAttributes=[typeid]:  Print all attributes of typeid.
    --PrintHelp:                 Print this help message.

salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run="prueba --dia=18"
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.196s)
Hoy es día 18 de marzo
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run="prueba --dia=27 --mes=septiembre"
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.201s)
Hoy es día 27 de septiembre
```

Figura 3-9: Terminal para ejemplo3.cc

3.6 Clase Ptr

Un puntero es un valor que especifica una dirección de memoria. El puntero siempre tiene un tipo asociado, que es el de la variable a la que apunta. Esta clase nos proporciona un puntero más inteligente (*smart pointer*). La diferencia entre este tipo de punteros de C++ y los punteros habituales es que el objeto se limpiará una vez que todas las instancias Ptr<> son destruidas.

Crear un puntero de este tipo se hace de la siguiente forma

```
Ptr<tipodevariable> nombre = CreateObject <tipodevariable> ();
```

Para acceder a los campos de la variable (atributos y métodos), no se realiza de la manera habitual con el signo '.' sino que, como debe hacerse a través de un puntero, será de la forma '->'.

3.7 Variables aleatorias

El ns3 proporciona una serie de variables que nos permite modelar distribuciones aleatorias.

Una variable aleatoria real se define como una función que asigna a cada elemento del espacio muestra un punto en la recta real. Se denota por $X(\alpha)$, y cumple las siguientes propiedades:

- $X(\alpha)$ es una función medible.
- La probabilidad de los sucesos $X(\alpha) = \pm \infty$ es igual a cero.

Por defecto, las simulaciones en ns3 utilizan una semilla fija, es decir, cada ejecución de una simulación generará resultados idénticos, a no ser que se modifique dicho parámetro.

3.7.1 Cambiar la semilla

Para ello tenemos a nuestra disposición la clase *RngSeedManager*, con las siguientes funciones:

GetNextStreamIndex(): devuelve el índice del siguiente flujo.

GetRun(): obtiene el número actual de la simulación.

GetSeed(): indica el número que funciona como semilla para todas las variables aleatorias.

SetRun(uint64_t *numero***)**: establece el número de la simulación.

SetSeed(uint32_t *numero***)**: hace que la semilla pase a ser el número establecido por dicha función.

Al modificar tanto un número como el otro, los valores de las variables aleatorias irán cambiando de forma pseudoaleatoria con ellos. Sus valores por defecto son igual a la unidad.

3.7.2 Variable Aleatoria Normal

Una variable aleatoria X sigue una distribución normal o de Gauss, $X \sim (\mu, \sigma^2)$, donde μ es la media y σ^2 la varianza, si su función densidad de probabilidad cumple que:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma}}, x \in \mathbb{R} \tag{3.1}$$

Esta distribución es ampliamente utilizada, porque, además de que multitud de experimentos en diversas ramas de la ciencia siguen esta distribución, bajo ciertas condiciones, la suma de variables aleatorias independientes sigue una distribución normal. Su función de densidad tiene la forma mostrada en la Figura 3-10. Si cambiamos la media, desplazaremos el centro de la gráfica al número escogido; si modificamos la varianza, afectará a la amplitud de la curva. Cuanto más pequeña sea la varianza, más estrecha será la función.

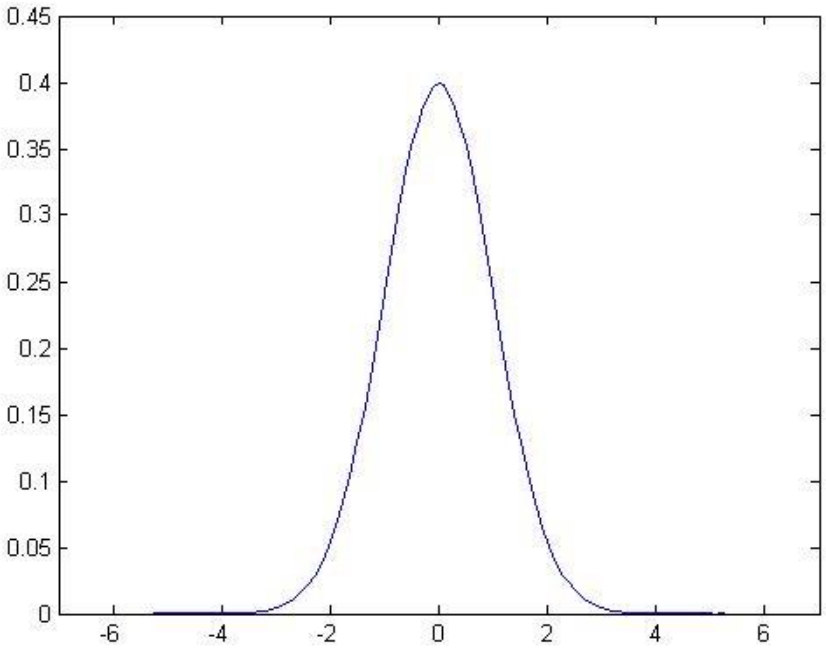

Figura 3-10: Función densidad de probabilidad de variable normal (0,1)

El ns3 nos proporciona tres atributos para poder variar las características de la distribución:

Mean: media de la distribución normal.

Variance: varianza de la distribución normal.

Bound: el intervalo en el que se moverá la variable aleatoria será [mean-bound, mean+bound].

Para poder realizarlo, crearemos un puntero a un objeto de tipo *NormalRandomVariable*. Para acceder a los atributos, se hará de la forma indicada en el apartado previo:

variable->**SetAttribute("***NombreAtributo***",** *NuevoValor***);**

Al introducir el nuevo valor, hay que especificar el tipo de variable que será, es decir, *DoubleValue(valor), TimeValue(valor)...*

Con el fin de obtener el valor de la variable aleatoria, usaremos la función *GetValue*.

```
#include <ns3/core-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
    double media = 1;
    double varianza = 0.1;
    double intervalo = 0.25;
    int i;


    Ptr<NormalRandomVariable> var =
                            CreateObject<NormalRandomVariable>();
    var->SetAttribute("Mean", DoubleValue(media));
    var->SetAttribute("Variance", DoubleValue(varianza));
    var->SetAttribute("Bound", DoubleValue(intervalo));

    for (i=0; i<=10; i++)
        std::cout << var->GetValue() << std::endl;

return 0;
}
```

Figura 3-11: Código de ejemplo4.cc

Como podemos comprobar en la terminal, obtenemos valores de una distribución normal con las características que le hemos especificado.

```
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run prueba
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (5.639s)
1.12844
0.971656
0.841882
1.05926
1.18275
1.05249
1.07452
0.842911
1.20795
0.771075
1.00754
salas@localhost:~/ns-allinone-3.26/ns-3.26$
```

Figura 3-12: Terminal para ejemplo4.cc

Se pueden realizar modificaciones con la semilla y el número de simulación de la forma nombrada anteriormente. Basta con añadir alguna de las siguientes sentencias antes de la creación de la variable aleatoria normal.

```
RngSeedManager::SetSeed(numero);
RngSeedManager::SetRun(numero);
```

```
#include <ns3/core-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
    double media = 1;
    double varianza = 0.1;
    double intervalo = 0.25;
    int i;

    RngSeedManager::SetSeed(2);
    Ptr<NormalRandomVariable> var =
                        CreateObject<NormalRandomVariable>();
    var->SetAttribute("Mean", DoubleValue(media));
    var->SetAttribute("Variance", DoubleValue(varianza));
    var->SetAttribute("Bound", DoubleValue(intervalo));

    for (i=0; i<=10; i++)
        std::cout << var->GetValue() << std::endl;

return 0;
}
```

Figura 3-13: Código de ejemplo4-2.cc

Como se puede comprobar, generará otra serie de valores con las mismas características que la distribución anterior, pero, debido a que hemos cambiado la semilla, son números diferentes.

3.7.3 Variable Aleatoria Uniforme

Una variable aleatoria X sigue una distribución uniforme en el intervalo [a, b] si su función de densidad tiene la siguiente expresión:

$$f(x) = \begin{cases} \frac{1}{b-a}, & si\ a \leq x \leq b \\ 0, & e.o.c. \end{cases} \tag{3.2}$$

El experimento asociado viene dado por cualquier experimento en el que la variable aleatoria toma valores reales en el intervalo [a, b] de forma igualmente probable.

Este software la modela como *UniformRandomVariable*, que define una variable que sigue una distribución uniforme. En este caso, los atributos que se pueden modificar son el valor mínimo y máximo donde se moverá la función, siendo el intervalo de la forma: [min, max).

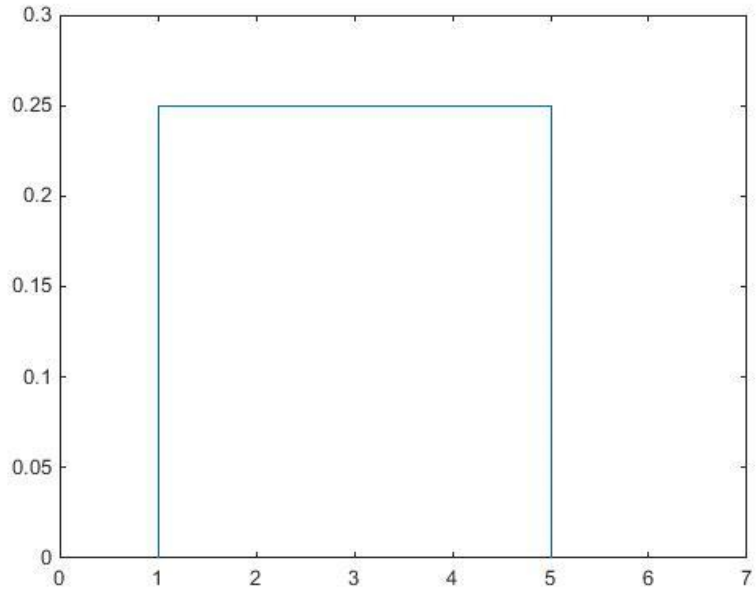

Figura 3-14: Función densidad de probabilidad de variable aleatoria [1, 4]

Si realizamos una simulación con el software con una variable uniforme entre los valores uno y cinco, obtendremos una serie de valores entre ambos, todos con igual probabilidad de aparecer. Como podemos comprobar en la Figura 3-16, los valores de dicha variable aleatoria están igualmente distribuidos a lo largo de todo el intervalo.

Además de estas variables aleatorias, en la documentación del ns3 podemos ver más, tales como *ExponentialRandomVariable, ErlangRandomVariable y GammaRandomVariable* entre otras.

```
#include <ns3/core-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
    double minimo = 1;
    double maximo = 5;
    int i;


    Ptr<UniformRandomVariable> var =
                          CreateObject<UniformRandomVariable>();
    var->SetAttribute("Min", DoubleValue(minimo));
    var->SetAttribute("Max", DoubleValue(maximo));

    for (i=0; i<=10; i++)
        std::cout << var->GetValue() << std::endl;

return 0;
}
```

Figura 3-15: Código de ejemplo5.c

```
salas@localhost:~/ns-allinone-3.26/ns-3.26$ ./waf --run prueba
Waf: Entering directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Waf: Leaving directory `/home/salas/ns-allinone-3.26/ns-3.26/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.291s)
4.26613
3.42738
2.9823
2.52637
2.47629
4.22644
1.40354
3.33918
4.75616
3.50442
4.2878
salas@localhost:~/ns-allinone-3.26/ns-3.26$
```

Figura 3-16: Terminal para ejemplo5.cc

3.8 Gráficas

El ns3 proporciona una clase que permite generar gráficas de tipo gnuplot a partir de un conjunto de datos. Podemos dividir las funcionalidades según fichero, gráfica y curva.

Para una gráfica, su clase es *Gnuplot*. De entre sus funciones podemos destacar:

AddDataset(GnuplotDataset *dato***)**: añade los datos de una curva a la gráfica.

SetTitle(String *Titulo***)**: inserta dicho título a la gráfica.

SetLegend(String *NombreEjeX***, String** *NombreEjeY***)**: nombra a la etiqueta de cada eje.

AppendExtra(String *extra***)**: añade órdenes extra de tipo gnuplot. Es especialmente relevante para controlar el tamaño de los ejes, pues, por defecto, se ajusta a los valores mínimos y máximos.

```
AppendExtra("set xrange [min:max]");
AppendExtra("set yrange [min:max]");
```

GenerateOutput(Ostream *fichero***)**: escribe los datos de la gráfica en el fichero de salida.

SetExtra(String *orden***)**: incluye otras órdenes de tipo gnuplot. Trabaja de la misma forma que *AppendExtra*, pero, gracias a la existencia de ambos métodos, podemos incluir dos órdenes extra. Esto se debe a que varias órdenes de tipo *SetExtra* o *AppendExtra* se sobreescriben entre ellas. Por ejemplo, una orden de tipo gnuplot que podemos resaltar es "*set key outside*", que sitúa la leyenda fuera de los márgenes de la gráfica.

Si añadimos varias curvas a una misma gráfica, a cada una de ellas se le asignará un color de forma automática.

Por otra parte, las curvas se definen con *Gnuplot2dDataset* y usaremos las siguientes funciones:

Add(Double *x***, Double** *y***)**: añade un punto (x,y) a la curva.

SetTitle(String *titulo***)**: nombra a la curva. Si no le ponemos un título, se llamará Untitled por defecto.

SetStyle(Gnuplot2dDataset::Enum Style *estilo***)**: se emplea cuando queremos variar el estilo de la gráfica. Las opciones son: LINES, POINTS, LINES_POINTS, DOTS, IMPULSES, STEPS, FSTEPS, HIPSTEPS, siendo el primero el estilo por defecto.

Para el caso del fichero, solo vamos a nombrar que se hace de la forma usual en C++:

Es de tipo **std:ofstream**, y, al llamar al constructor, se le pasa el nombre de dicho fichero como *string*, que, en

este caso, debe tener la extensión .plt (para que sea un archivo gnuplot).

Con la función **close()** cerramos dicho fichero una vez que lo hayamos escrito.

Con las clases y funciones mostradas, obtendremos un fichero de tipo gnuplot. Para poder visualizar mejor el resultado, podemos generar la gráfica en formato imagen, por ejemplo, tipo .png. Con este fin, al llamar al constructor de la gráfica, debemos pasarle como parámetro el nombre que queremos darle al archivo imagen ("imagen.png"). Tras llevar a cabo la simulación con nuestro programa ns3, deberemos ejecutar en el terminal el archivo gnuplot que se ha generado para obtener la imagen, de la forma:

```
gnuplot fichero.plt
```

Vamos a realizar una gráfica partiendo del ejemplo3.cc.

```
#include <ns3/core-module.h>
#include <ns3/gnuplot.h>

using namespace ns3;

int main(int argc, char*argv[])
{
    double media = 1;
    double varianza = 0.1;
    double intervalo = 0.25;
    double valor;
    int i;

    Gnuplot grafica("ejemplo6.png");
    grafica.SetTitle("ejemplo6.cc");
    grafica.SetLegend("Muestras", "Variable aleatoria normal");
    grafica.AppendExtra ("set xrange [-1:11]");
    grafica.AppendExtra("set yrange [0:1.5]");

    Gnuplot2dDataset curva;
    curva.SetTitle("Variable Normal");
    curva.SetStyle(Gnuplot2dDataset::LINES_POINTS);


    Ptr<NormalRandomVariable> var =
                          CreateObject<NormalRandomVariable>();
    var->SetAttribute("Mean", DoubleValue(media));
    var->SetAttribute("Variance", DoubleValue(varianza));
    var->SetAttribute("Bound", DoubleValue(intervalo));

    for (i=0; i<=10; i++)
    {
        valor=var->GetValue();
        std::cout << valor << std::endl;
        curva.Add(i, valor);
    }

    grafica.AddDataset(curva);
    std::ofstream fichero ("ejemplo6.plt");
    grafica.GenerateOutput (fichero);
    fichero.close ();


return 0;
}
```

Figura 3-17: Código de ejemplo6.cc

Si ejecutamos la orden antes mencionada para obtener ejemplo6.png, tendremos la siguiente gráfica:

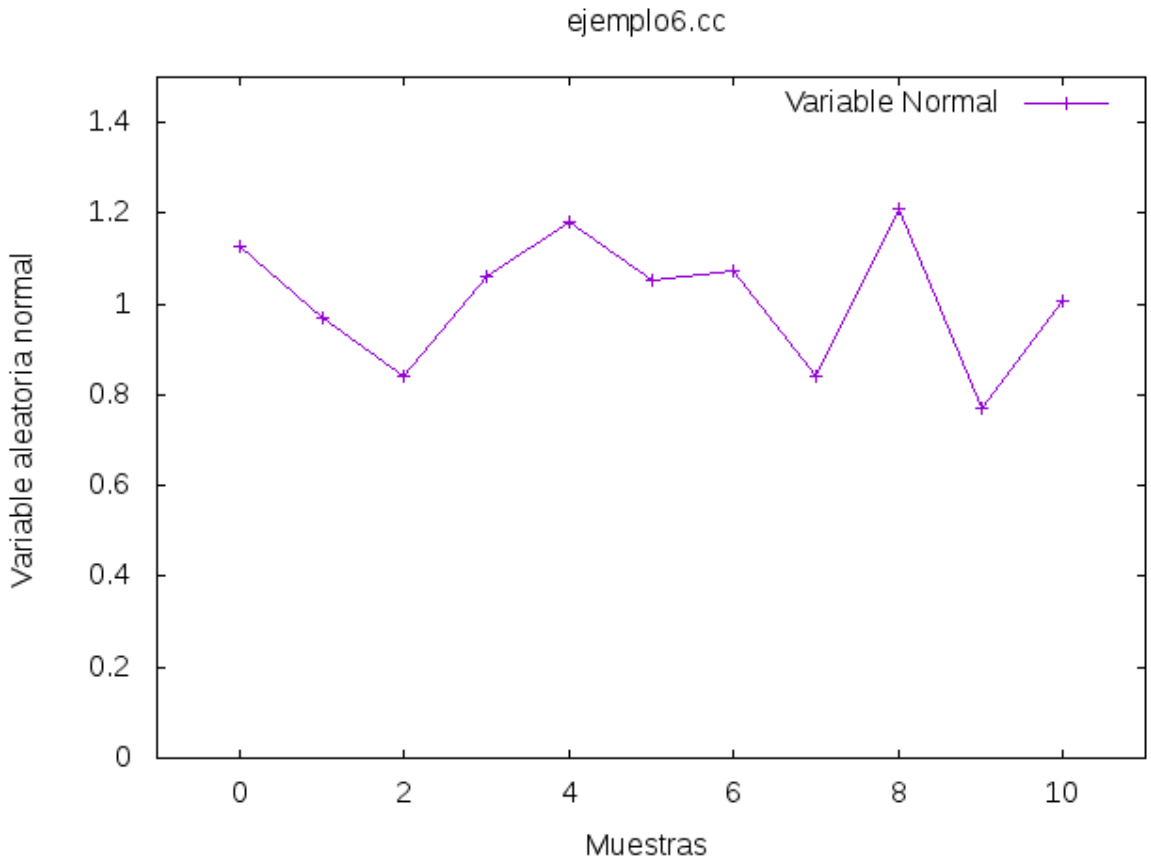

Figura 3-18: Imagen generada por ejemplo6.cc

3.9 Time

Como ya hemos mencionado antes, nos proporciona una variable de tipo tiempo, que se puede expresar en varias unidades. Al llamar al constructor lo hacemos de la forma:

Time(String *NumeroUnidad***)**, por ejemplo: Time("10ms"), sin espacio entre el número y la unidad. Las unidades que hay son:

- y (years, años)
- d (days, días)
- h (hours, horas)
- min (minutes, minutos)
- s (seconds, segundos)
- ms (milliseconds, milisegundos)
- us (microseconds, microsegundos)
- ns (nanoseconds, nanosegundos)
- ps (picoseconds, picosegundos)
- fs (femtoseconds, femtosegundos)

Por otra parte, podemos destacar las siguientes funciones:

FromInteger(uint64_t *numero***, enum Unit** *unidad***)**: crea una variable tipo *Time* a partir de un entero, con valor *numero* y la unidad especificada.

FromDouble(double *numero***, enum Unit** *unidad***)***:* funciona igual que la anterior, pero a partir de un tipo double.

ToInteger(enum Unit *unidad***)**: convierte una variable tipo *Time* a un entero (uint64_t) en una unidad particular.

ToDouble(enum Unit *unidad***)**: del mismo modo que el anterior, convierte a un tipo double.

GetInteger(): convierte a tipo entero en la unidad de resolución establecida.

GetDouble(): igual que el anterior.

Aunque existen funciones específicas como *Compare*(Time *valor*) para comparar dos tiempos distintos, se pueden utilizar los operadores habituales con dos tiempos, tales como +, -, +=, -=, ==, !=, <, >, <=, >=.

3.10 DataRate

Es una clase para especificar velocidades de transmisión de datos. Especificar una velocidad consiste en:

- Un valor numérico.
- Un prefijo multiplicador opcional:
  - k, K: 1000.
  - Ki: 1024.
  - M: 10^6.
  - Mi: 1024 Ki.
  - G: 10^9.
  - Gi: 1024 Mi
- Una unidad:
  - b: bits.
  - B: bytes.
  - ps, /s: por segundo.

En este caso, puede haber un espacio a continuación del número, pero no entre el prefijo y la unidad. Por ejemplo, velocidades válidas son: “128kbps” y “64 b/s”.

Algunas funciones destacadas son:

GetBitRate(): devuelve la velocidad en bps.

CalculateBitsTxTime(uint32_t *numero***)**: proporciona el tiempo que tarda en transmitir *numero* bits con la tasa de velocidad que tiene.

CalculateBytesTxTime(uint32_t *numero***)**: igual que el anterior, pero introduciendo bytes.

Al igual que la clase *Time*, tiene funciones propias para la comparación de varias velocidades, aunque pueden utilizarse las siguientes operaciones: <, >, <=, >=, ==, !=.

3.11 Eventos

3.11.1 EventId

Identifica un evento único en una simulación, programado mediante el *Schedule*.

EventId(): es el constructor por defecto.

EventId(Ptr<EventImpl> *evento***, uint64_t** *tiempo***, uint32_t** *contexto***, uint32_t** *identificador***)**: es el constructor específico de la clase. Especificamos la marca de tiempo, el contexto y el identificador correspondiente.

Cancel(): cancela el evento, al igual que *Simulator::Cancel*.

GetContext(): devuelve el contexto en el que se encuentra dicho evento.

GetTs(): proporciona el tiempo en el que expira el evento (asignado mediante *Schedule*).

GetUid(): indica el identificador único de dicho evento.

IsExpired(): devuelve *true* si el evento ya ha concluido, al igual que la función del mismo nombre de la clase *Simulator*.

IsRunning(): es la función negada de la anterior.

PeekEventImpl(): devuelve un puntero a la función que se llama cuando se ejecuta el evento.

Existen otras funciones para comparar si los identificadores de dos eventos son iguales o distintos, pero también se pueden aplicar los operadores == y !=.

3.11.2 EventImpl

Representa un evento de simulación, es decir, la función que se ejecutará cuando se programe *Schedule*.

Invoke(): es un método llamado por la clase *Schedule* cuando se alcanza el tiempo establecido para dicho evento.

Cancel(): no se elimina de la lista de eventos, pero cambia su estado a *cancelado*, el cual se mira siempre que se llama a la función *Invoke*.

IsCancelled(): será *true* si el evento asociado ha sido cancelado.

Para crear instancias de dicha clase a partir de punteros a funciones, podemos hacer uso de la siguiente función, que pertenece al núcleo de la programación de ns3:

MakeEvent(void * *función***)**: este método devolverá un puntero de tipo *EventImpl*, que podrá ser usado para programar eventos que concluyan con la ejecución de la *función.* Si esta función necesita que se le pasen parámetros, se hará en los siguientes argumentos, hasta un número máximo de seis. A partir de modificaciones del código de ejemplo1.cc, hemos creado el siguiente ejemplo, para ilustrar este apartado:

```
#include <ns3/core-module.h>

using namespace ns3;

void funcion(Time tiempo);

int main(int argc, char*argv[])
{
    Time temp = Time("10s");

    Ptr<EventImpl> evento = MakeEvent(&funcion, temp);

    Simulator::Schedule(temp, evento);
    Simulator::Run();
    Simulator::Destroy();

return 0;
}

void funcion(Time tiempo)
{
    std::cout << "Función llamada en: " << tiempo.GetSeconds()
              << "s" << std::endl;
}
```

Figura 3-19: Código de ejemplo7.cc

3.12 Packet

Esta clase nos permite crear paquetes que enviaremos entre dos nodos o dispositivos para simular una transmisión real. Tenemos tres formas de hacer un paquete:

- Si no nos importa el contenido del paquete, podemos rellenarlo de ceros, con el tamaño que le especifiquemos.
- Podemos crear un paquete con el contenido que le detallemos para que lo guarde en un *buffer* interno.
- Podemos añadir una cabecera o una cola con el contenido que queramos a un paquete ya existente.

3.12.1 Crear paquete (1)

En el primer caso, tan solo tenemos que llamar al constructor de dicha clase, con *tamaño* en bytes:

Ptr<Packet> *paquete* **= Create<Packet>(uint32_t** *tamaño***);**

También se puede crear un paquete vacío, simplemente llamando al constructor sin indicar tamaño alguno.

3.12.2 Crear paquete (2)

Para la segunda opción, primero debemos crear una cadena de caracteres con el contenido, y copiarlo en el *buffer* interno del paquete para enviarlo.

Ptr<Packet> *paquete* **= Create<Packet>(uint8_t*** *buffer* **, uint32_t** *tamaño***);**

En el siguiente ejemplo, simulamos la parte del transmisión y recepción del paquete, suponiendo que se ha mandado por el canal correspondiente:

```
#include <ns3/core-module.h>
#include <ns3/packet.h>

using namespace ns3;

int main(int argc, char*argv[])
{
    /* Transmisión */

    uint8_t buffer = 255;

    Ptr<Packet> paquete = Create<Packet> (&buffer, 1);


    /* Recepción */

    uint8_t contenido=0;

    paquete->CopyData(&contenido, paquete->GetSize());

    std::cout << "Recibido: " << (int)contenido << std::endl;

return 0;
}
```

Figura 3-20: Código de ejemplo8.cc

Para este ejemplo hemos necesitado dos métodos de esta clase:

GetSize(): nos devuelve el tamaño en bytes del paquete al que se refiere.

CopyData(uint8_t *buffer***, uint32_t** *tamaño***)**: copia el contenido del paquete al *buffer* especificado. El tamaño que hay que proporcionarle es el del propio *buffer*.

3.12.3 Crear paquete (3)

La tercera y última manera de crear un paquete es añadir contenido a un paquete ya creado. En este caso, instanciamos el paquete del mismo modo que la primera opción y, a continuación, le añadimos una cabecera con los datos que queramos (número del 0 al 65535, pues los datos tienen formato uint16_t).

Para poder realizarlo, necesitamos la clase *MyHeader*, que no está definida como una de las clases del software, pero, sin embargo, la documentación nos proporciona el código siguiente para utilizarla:

```
#include "ns3/ptr.h"
#include "ns3/packet.h"
#include "ns3/header.h"
#include <iostream>

using namespace ns3;

/* A sample Header implementation
*/
class MyHeader : public Header
{
public:

  MyHeader ();
  virtual ~MyHeader ();

  void SetData (uint16_t data);
  uint16_t GetData (void) const;

  static TypeId GetTypeId (void);
  virtual TypeId GetInstanceTypeId (void) const;
  virtual void Print (std::ostream &os) const;
  virtual void Serialize (Buffer::Iterator start) const;
  virtual uint32_t Deserialize (Buffer::Iterator start);
  virtual uint32_t GetSerializedSize (void) const;
private:
  uint16_t m_data;
};


MyHeader::MyHeader ()
{
  // we must provide a public default constructor,
  // implicit or explicit, but never private.
}
MyHeader::~MyHeader ()
{
}

TypeId
MyHeader::GetTypeId (void)
{
 static TypeId tid = TypeId ("ns3::MyHeader")
   .SetParent<Header> ()
    .AddConstructor<MyHeader> ()
  ;
  return tid;
}
TypeId
MyHeader::GetInstanceTypeId (void) const
{
  return GetTypeId ();
}

void
MyHeader::Print (std::ostream &os) const
{
  // This method is invoked by the packet printing
  // routines to print the content of my header.
  //os << "data=" << m_data << std::endl;
  os << "data=" << m_data;
}
```

```
uint32_t
MyHeader::GetSerializedSize (void) const
{
  // we reserve 2 bytes for our header.
  return 2;
}
void
MyHeader::Serialize (Buffer::Iterator start) const
 {
   // we can serialize two bytes at the start of the buffer.
   // we write them in network byte order.
   start.WriteHtonU16 (m_data);
 }
uint32_t
MyHeader::Deserialize (Buffer::Iterator start)
{
  // we can deserialize two bytes from the start of the buffer.
  // we read them in network byte order and store them
  // in host byte order.
  m_data = start.ReadNtohU16 ();

  // we return the number of bytes effectively read.
  return 2;
}


void
MyHeader::SetData (uint16_t data)
{
  m_data = data;
}
uint16_t
MyHeader::GetData (void) const
{
  return m_data;
}
```

Figura 3-21: Código de la clase *MyHeader*

Gracias a esta clase, podemos agregar cabeceras a un paquete con la siguiente función en el transmisor:

SetData(uint16_t *contenido***)**: añadimos a la cabecera los datos que queremos enviar.

Uno de los métodos propios de la clase *Packet* es necesario también para poder realizar la transmisión:

AddHeader(Header *cabecera***)**: se emplea para incluir la cabecera especificada en el paquete a enviar. La clase *Header* está definida dentro del software ns3, aunque solo contiene los métodos generales de *MyHeader*. Para añadir cabeceras sin datos, podemos emplear la primera, pero si queremos personalizar las cabeceras introduciendo contenido, es necesario definir la clase *MyHeader*. Para el uso de este método, podemos emplear la clase definida previamente, ya que, como podemos observar, *MyHeader* es una implementación de la clase *Header*, pero añadiendo más funciones y atributos.

Por otra parte, en el receptor será necesario:

RemoveHeader(Header *cabecera***)**: devuelve la cabecera del paquete en la dirección que le pasemos como parámetro. Al igual que el método anterior, pertenece a la clase *Packet* y podemos utilizar la cabecera *MyHeader* definida anteriormente.

Con el fin de imprimir los datos que se hayan transmitido en la cabecera del paquete, tenemos dos opciones:

GetData(): obtenemos el contenido de la cabecera y, posteriormente, procedemos a imprimirlo con los métodos habituales.

Print(std::ostream *BufferDeSalida***)**: es un método que imprime el contenido de la cabecera. Para realizarlo de la forma recurrente hasta ahora, el parámetro que debemos pasarle será *std::cout*. Debemos tener en cuenta que este tipo de impresión no termina con un salto de línea.

Podemos comprobar todo esto en el siguiente ejemplo, en el que tenemos definida también la clase *MyHeader,* anteriormente mencionada:

```
#include <ns3/core-module.h>
#include <ns3/packet.h>

using namespace ns3;

int main(int argc, char*argv[])
{
  /* Transmisión */

  Ptr<Packet> paquete = Create<Packet> (1);
  MyHeader cabecera = MyHeader();
  cabecera.SetData (27913);
  paquete->AddHeader (cabecera);

  /* Recepción */
  MyHeader recibido = MyHeader();

  paquete->RemoveHeader(recibido);

  recibido.Print(std::cout);
  std::cout << "\nRecibido: " << recibido.GetData () << std::endl;

return 0;
}
```

Figura 3-22: Código de ejemplo9.cc

3.12.4 Aspectos generales

Además de los métodos de la clase *Packet* mencionados hasta ahora, vamos a destacar los siguientes:

AddAtEnd(Ptr<Packet> *paquete***)**: concatena el paquete actual al final del paquete que le indiquemos.

AddPaddingAtEnd(uint32_t *tamaño***)**: añade un relleno de ceros de dicho tamaño al final del paquete.

CreateFragment(uint32_t *principio***, uint32_t** *longitud***)**: crea un fragmento del paquete original a partir del byte *principio* del paquete original, con la longitud especificada.

EnablePrinting(): por defecto, los paquetes no tienen habilitada la opción de imprimir su contenido. Para que el método *Print* funcione, se debe llamar primero a esta función. Si se quiere habilitar para todos los paquetes, basta con poner la sentencia siguiente antes de crear ningún paquete:

```
Packet::EnablePrinting()
```

GetUid(): devuelve el identificador único del paquete. Cabe destacar que este identificador es a nivel interno del programa, de tal manera que no tiene ninguna equivalencia con un número de secuencia.

Print(std::ostream *BufferDeSalida***)**: imprime los datos del paquete, incluyendo la cabecera si la hay. Hay que tener en cuenta que no imprime directamente el contenido del paquete, sino las características de este. Recordar que es necesario habilitar la impresión antes de llamar a esta función.

RemoveAtEnd(uint32_t *tamaño***)**: elimina el número de bytes que le señalemos del final del paquete. El software nos permite que el tamaño que indiquemos sea mayor que el del paquete, eliminando asi todos los bytes del paquete.

RemoveAtStart(uint32_t *tamaño***)**: ídem al anterior, pero con respecto a los primeros bytes del paquete.

ToString(): convierte a una cadena de caracteres los datos correspondientes al paquete. En este caso, también se debe habilitar la impresión; en caso contrario, tendremos una cadena en blanco. Imprimir dicha cadena por pantalla tendrá el mismo efecto que llamar al método *Print*.

Copy(): devuelve un paquete con el mismo contenido que el paquete anterior. Lo habitual es llamar a esta función al crear el nuevo paquete:

```
Ptr<Packet> paqueteNuevo = paqueteAntiguo->Copy();
```

Es necesario mencionar que, tanto a los paquetes como a los bytes individuales se les puede asociar una etiqueta (*tag*) que los identifique con algunos métodos que proporciona esta clase. Para más información, consulte la documentación de este software.

3.12.5 Número de secuencia

Un número de secuencia es un identificador del flujo de datos, es decir, de un paquete. Estos números son consecutivos para paquetes que también lo son. Dependiendo del protocolo de encapsulación del paquete, el número de secuencia puede tener distinta longitud. Al llegar al número máximo, el siguiente paquete empezará otra vez la secuencia, por lo que se le asignará el valor cero.

Este software nos proporciona una clase *SequenceNumber* que nos genera los números de secuencia correspondientes. Nos permite secuencias de 8 (hasta 255), 16 (hasta 65.535) y 32 bits (hasta 4.294.967.295). Para usar esta clase es necesaria la siguiente plantilla:

```
SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE>;
```

El primero de los parámetros, *NUMERIC_TYPE,* corresponde a uno de los tipos de enteros sin signo de la librería <*cstdin*>, es decir, uint8_t, uint16_t y uint32_t. Por su parte, el segundo argumento será su correspondiente tipo con signo, es decir, int8_t, int16_t e int32_t. Para facilitar el uso de dicha plantilla, ns3 nos proporciona unos tipos definidos que equivalen a ellos:

Tabla 3-2. Equivalencias en la clase *SequenceNumber*

SequenceNumber<uint8_t, int8_t>	SequenceNumber8
SequenceNumber<uint16_t, int16_t>	SequenceNumber16
SequenceNumber<uint32_t, int32_t>	SequenceNumber32

Tenemos tres constructores diferentes para esta clase:

SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>()**: constructor por defecto de la clase, tomará el valor cero.

SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>(NUMERIC_TYPE** *valor***)**: el primer número de secuencia es el valor que le proporcionemos.

SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>(SequenceNumber<**NUMERIC_TYPE, SIGNED_TYPE**>** *secuencia***)**: construye una secuencia a partir de otra dada.

Para obtener el valor de una secuencia en un determinado instante, recurriremos al método **GetValue()**. El número correspondiente tiene el formato sin signo del número de secuencia asignado.

```
#include <ns3/core-module.h>
#include <ns3/sequence-number.h>

using namespace ns3;


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

    uint16_t dato=3;

    SequenceNumber<uint16_t,int16_t> secuencia =
                            SequenceNumber<uint16_t,int16_t>(dato);
    SequenceNumber16 secuencia2 = SequenceNumber16(secuencia);

    std::cout << "Secuencia 1: " << (int)secuencia.GetValue()
              << std::endl;
    std::cout << "Secuencia 2: " << (int)secuencia2.GetValue()
              << std::endl;

return 0;
}
```

Figura 3-23: Código de ejemplo10.cc

Las operaciones aritméticas con números de secuencia son complejas, debido a que empieza por cero una vez ha terminado la secuencia. Por ello, la clase *SequenceNumber* nos proporciona una serie de funciones, específicamente diseñadas para este tipo de operaciones:

operator!=(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: compara dos secuencias, devolviendo *true* si son distintas.

operator+(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: suma dos secuencias y devuelve la suma convertida en otro número de secuencia.

operator+(SIGNED_TYPE *valor***)**: suma el valor indicado al número de secuencia. Al igual que el anterior, devuelve dicha suma en un número de secuencia.

operator++(): incrementa el número de secuencia en una unidad. Es un operador de tipo prefijo (*prefix*) por lo que, al llamar a la función, se incrementa automáticamente.

operator++(int *valor***)**: incrementa el número de secuencia en una unidad. A diferencia del anterior, es un operador de tipo sufijo (*postfix*), es decir, al ejecutar la función tendrá el valor previo, y, en la siguiente operación, será cuando ya esté incrementado. El valor que le pasamos por parámetro solo sirve para diferenciar ambos métodos (polimorfismo).

operator+=(SIGNED_TYPE *valor***)**: incrementa el número de secuencia en el valor que le indiquemos.

operator-(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: realiza la resta entre las dos secuencias y devuelve la diferencia en otro número de secuencia.

operator-(SIGNED_TYPE *valor***)**: resta el valor proporcionado al número de secuencia. Al igual que el anterior, devuelve el resultado en otra secuencia.

operator- -(): decrementa el número de secuencia en una unidad. Es de tipo prefijo.

operator- -(int *valor***)**: decrementa el número de secuencia en una unidad. Al contrario que el anterior, es de tipo sufijo.

operator-=(SIGNED_TYPE *valor***)**: decrementa el número de secuencia en el valor que le indiquemos.

operator<(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: compara dos secuencias. Si la primera es menor, devuelve el booleano *true.*

operator<=(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: compara si el primer número de secuencia es menor o igual que el otro. En caso afirmativo, devuelve *true*.

operator==(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: verifica si hay igualdad entre dos secuencias y devuelve un tipo bool.

operator>(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: devolverá *true* si la otra secuencia es mayor.

operator>=(SequenceNumber<NUMERIC_TYPE, SIGNED_TYPE**>** *OtraSecuencia***)**: compara dos secuencias. Si la segunda es mayor o igual que la primera, devolverá un booleano afirmativo.

En el siguiente ejemplo se puede comprobar el uso de algunas de estas funciones.

```
#include <ns3/core-module.h>
#include <ns3/sequence-number.h>

using namespace ns3;


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

    uint8_t dato = 3;

    SequenceNumber<uint8_t,int8_t> secuencia =
                                SequenceNumber<uint8_t,int8_t>();
    SequenceNumber8 auxiliar;

    std::cout << "Valor inicial de la secuencia: "
              << (int)secuencia.GetValue() << std::endl;

    auxiliar=secuencia.operator+(dato);
    std::cout << "Valor de la secuencia: "
              << (int)secuencia.GetValue();
    std::cout << ", resultado de sumar con " << (int)dato << " es: "
              << (int)auxiliar.GetValue()<< std::endl;

    std::cout << "Valor de la secuencia: "
              << (int)secuencia.GetValue();
    std::cout << ", incrementar con prefijo: "
              << (int)secuencia.operator++().GetValue() << std::endl;

    std::cout << "Valor de la secuencia: "
              << (int)secuencia.GetValue();
    std::cout << ", incrementar con sufijo: "
              << (int)secuencia.operator++(dato).GetValue();
    std::cout << ", valor a posteriori: "
              << (int)secuencia.GetValue() << std::endl;


    if (secuencia.operator>(auxiliar))
        std::cout
          << "La secuencia original es mayor que la auxiliar"
          << std::endl;
    else
        std::cout
          << "La secuencia original es menor o igual que la auxiliar"
          << std::endl;


return 0;
}
```

Figura 3-24: Código de ejemplo10-2.cc

El uso de números de secuencia es especialmente relevante en transmisiones donde se necesita retransmisión. Cada paquete se distingue de los demás gracias a dicho número, y se puede volver a enviar un paquete con la misma información, debido a que sabemos el número de secuencia que se ha perdido.

4 SISTEMA DE COMUNICACIONES

Tell me and I forget. Teach me and I remember. Involve me and I learn.

Dime y lo olvido, enséñame y lo recuerdo, involúcrame y lo aprendo."

- Benjamin Franklin -

La comunicación consiste en intercambiar información mediante un código común entre un transmisor y un receptor. Esto puede corresponder a algo tan sencillo como un simple diálogo cara a cara entre dos personas, hasta algo tan complejo como una transmisión con el robot *Curiosity* situado en el planeta Marte, a varios millones de kilómetros de distancia.

Todas las señales de nuestro alrededor son analógicas (audio, vídeo…), excepto las producidas por sistemas digitales, como, por ejemplo, los ordenadores.

Las señales analógicas son generadas por fenómenos electromagnéticos y describen una onda continua que va variando en función del tiempo; esto implica que hay infinitos valores de dicha forma de onda.

Las señales digitales, sin embargo, son discretas y tienen un número finito de formas de onda. Esto favorece que la transmisión de cada unidad mínima de información (a la que llamaremos bit) sea independiente de la magnitud física a la que represente. Una de las grandes ventajas de este tipo de comunicaciones es la regeneración, la cual consiste en reconstruir una señal en un punto intermedio de la comunicación, de modo que se vuelve a transmitir en dicho punto sin influencia de ruido. Esto no es posible en comunicaciones analógicas en las cuales se dispone solo de repetidores que amplifican tanto la señal como el ruido.

Por tanto, el objetivo de un sistema de comunicación digital es la estimación, al final del trayecto, de la señal enviada y transmitida por un medio físico, con una calidad y fidelidad suficiente.

4.1 Elementos de un sistema de comunicaciones digitales

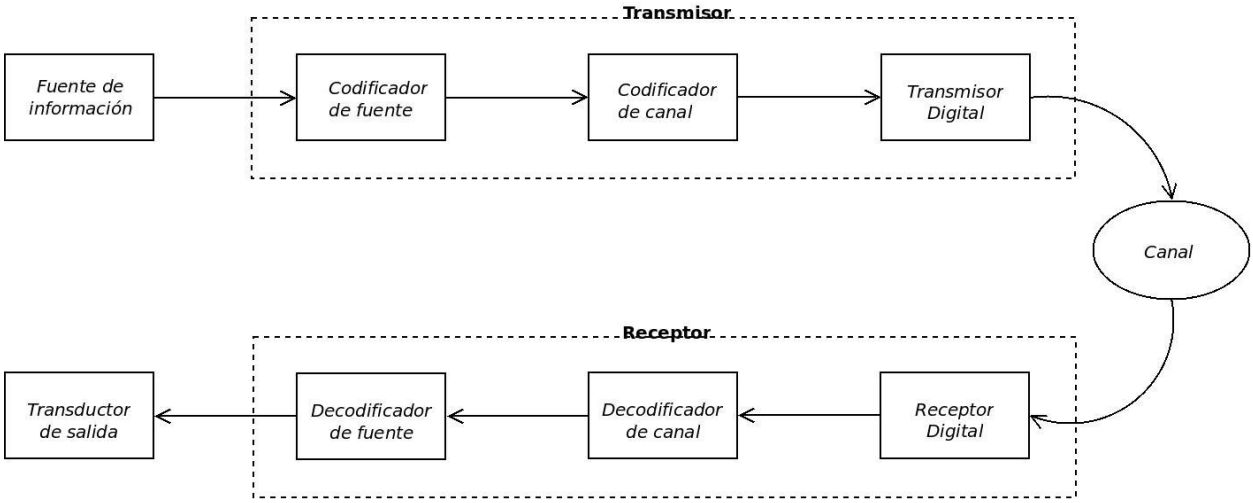

Figura 4-1: Esquema de un sistema de comunicaciones digitales.

La Figura 4-1 muestra el esquema típico con los elementos básicos de un sistema de comunicaciones digitales, en el que hay un solo transmisor de información y un único usuario.

La *fuente de información* es la que genera la señal a la que llamaremos mensaje, ya sea analógica o digital.

El bloque *codificador de fuente* se encarga de convertir el mensaje en una secuencia de dígitos binarios (bits), a la cual nos referiremos como secuencia de información. Su objetivo es representar el mensaje con el mínimo número de dígitos binarios posibles, eliminando cualquier redundancia que pueda tener, para conseguir una representación eficiente. A este proceso se le llama también *compresión de datos.* Lógicamente, cuanto mayor redundancia tuviese el mensaje, mayor eficiente será la codificación.

El *codificador de canal* se ocupa de proteger la secuencia de información contra los errores que se producirán en el canal, el cual explicaremos posteriormente. Con dicho fin, introducirá de manera controlada redundancia en la secuencia. El funcionamiento de este elemento puede parecer contrario al efecto del bloque anterior, pero se debe tener en cuenta que, al eliminar toda redundancia en el codificador de fuente, cada bit será más relevante, por lo que recibirlo de forma segura se hace fundamental. Una forma sencilla pero poco eficiente de hacerlo, sería repetir cada bit un número *n* de veces.

La secuencia de dígitos binarios obtenida tras el elemento anterior debe ser ahora transformada en una forma de onda que se adapte a las características del canal. Es por ello que el *transmisor digital* lleva a cabo un proceso de modulación, que consiste en modificar la frecuencia, amplitud o fase de distintas formas de onda para transmitir diversos niveles digitales. La clave de un buen diseño consiste en disponer del mayor número posible de formas de onda, pero suficientemente distintas, para que, al ser deterioradas en el canal, se puedan seguir distinguiendo en el receptor.

El *canal* se corresponde con el medio físico empleado para mandar la señal tras el proceso de modulación. Se puede modelar de diversas formas, pero su característica principal es que deteriora la señal de manera aleatoria, debido a mecanismos tales como retrasos, interferencias, atenuaciones…

Una vez la señal llega al receptor, el *receptor digital* realiza dos operaciones. La primera, a la que llamamos *demodulador digital*, se encarga de obtener la máxima información posible sobre la onda recibida. Por otra parte, la segunda operación la realiza el *detector*, que, a partir de la información recibida del demodulador, realiza en una estimación de qué símbolo fue transmitido de acuerdo a una regla preestablecida.

A continuación, se encuentra el *decodificador de canal*, que realizará el mecanismo contrario al codificador, es decir, conociendo el código empleado en el transmisor, intentará reconstruir la secuencia original de información.

Por último, el *decodificador de fuente* reconstruirá la forma de onda generada por la fuente de información, teniendo a su disposición las características del codificador correspondiente. Este mensaje recibido será una aproximación al enviado, y será entregado al *transductor de salida*, que es el destino final de dicha comunicación.

En aplicaciones con una comunicación bidireccional, lo habitual es encapsular el transmisor y receptor en un único equipo denominado *módem* (modulador-demodulador).

4.2 Esquema simplificado

El diagrama del apartado anterior puede reducirse al esquema de la Figura 4-2. Consiste en un inyector de información, un transmisor, un canal y un receptor. Para llevar a cabo la simulación en el software correspondiente, tendremos en cuenta dicho diagrama.

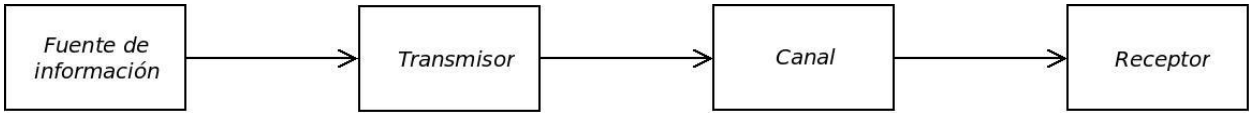

Figura 4-2: Esquema simplificado de un sistema de comunicaciones digitales.

El diseño que vamos a realizar está basado en dispositivos punto a punto, por lo que haremos hincapié en las características de los elementos que conforman una comunicación de dicho tipo.

4.3 Nodo

Para diseñar cualquier sistema de comunicaciones necesitaremos sus elementos fundamentales. En este caso, el elemento básico es el nodo. Cada punto de la red donde se transmiten o reciben mensajes es un nodo. En ns3, la clase correspondiente está definida en la cabecera *node.h*.

Esta clase permite unir:

- Una lista de objetos dispositivos de red (*NetDevice*), que representan las interfaces de red de cada nodo que están conectadas a otros nodos a través de canales.
- Una lista de objetos de aplicación, que representan las aplicaciones de generación de tráfico de usuario, y que interaccionan con el nodo a través del socket API (interfaz de programación de aplicaciones).
- Un identificador único de cada nodo (*node Id*).
- Un identificador de sistema, para simulaciones paralelas.

Cada nodo que creemos se añade a la lista de nodos automáticamente (*NodeList*), que contendrá la lista de todos los nodos empleados en la simulación, empezando por el cero.

Entre sus funciones principales, vamos a destacar las siguientes:

AddApplication(Ptr<Application> *aplicacion***)**: añade la aplicación previamente definida a dicho nodo.

AddDevice(Ptr<NetDevice> *dispositivo***)**: asocia un dispositivo de red concreto a dicho nodo.

GetId(): devuelve el identificador del nodo actual.

GetSystemId(): devuelve el identificador del proceso asociado a este nodo.

GetNApplications(): nos indica el número de aplicaciones asociadas a dicho nodo.

GetNDevices(): funciona del mismo modo que el método anterior, pero, en vez de aplicaciones, cuenta dispositivos.

PromiscReceiveFromDevice(Ptr<NetDevice> *dispositivo***, Ptr<Packet>** *paquete***, uint16_t** *protocol***, Address** *transmisor***, Address** *destino***, NetDevice::PacketType** *TipoDePaquete***)**: su objetivo es recibir paquetes de dispositivos que están en modo promiscuo. Este modo es aquel en el que el dispositivo captura todo el tráfico que pasa por su red, aunque los paquetes no tengan su dirección como destino.

NonPromiscReceiveFromDevice(Ptr<NetDevice> *dispositivo***, Ptr<Packet>** *paquete***, uint16_t** *protocol***, Address** *transmisor***)**: se emplea para recibir paquetes de dispositivos que están en modo no promiscuo.

ReceiveFromDevice(Ptr<NetDevice> *dispositivo***, Ptr<Packet>** *paquete***, uint16_t** *protocol***, Address** *transmisor***, Address** *destino***, NetDevice::PacketType** *TipoDePaquete***, bool** *promiscuo***)**: su finalidad es recibir un paquete de un dispositivo concreto.

4.4 NetDevice

Define la capa de red, es decir, el tipo de dispositivo en el que consiste. Tenemos, por ejemplo, *PointToPointNetDevice, WifiNetDevice, WimaxNetDevice, CSMANetDevice…*, dependiendo del tipo de red que vayamos a usar, habrá que emplear un dispositivo u otro.

En general, sus funciones públicas son las mismas, exceptuando algunas particulares según su uso. De estas, vamos a resaltar la función *send*, que sirve para enviar paquetes con un destino y un protocolo determinados. Se encarga de notificar a las capas superiores para que manden el paquete al destino correspondiente.

Send(Ptr<Packet> *paquete* **, const Address** *destino***, uint16_t** *protocolo***);**

El protocolo correspondiente dependerá del tipo de dispositivo que conectemos. Podemos diferenciar los siguientes métodos:

GetAddress(): obtiene la dirección de la interfaz actual.

GetChannel(): devuelve el canal al que está conectado dicho dispositivo. En el caso de que no haya ninguno asociado, devolverá cero.

GetNode(): nos proporciona el nodo al que está asociado este dispositivo de red.

IsBridge(): indica si el elemento de red está actuando como un puente.

IsLinkUp(): comprueba si el enlace está listo.

IsPointToPoint(): devuelve *true* si el enlace es de tipo punto a punto.

NeedsArp(): llama a las capas superiores para saber si es necesario activar el protocolo ARP.

SendFrom(Ptr<Packet> *paquete***, Address** *origen***, Adress** *destino***, uint16_t** *protocolo***)**: es similar al primer método explicado, pero especificando también la dirección origen del paquete.

SetAddress(Adress *dirección***)**: sirve para establecer la dirección de la interfaz.

SetNode(Ptr<Node> *nodo***)**: esta función se llama desde *Node::AddDevice*, para vincular el nodo y el dispositivo de red.

4.4.1 PointToPointNetDevice

Todos los dispositivos de red heredarán los atributos y métodos de la clase *NetDevice*, pues todas las demás clases serán subclases de esta. En el caso particular de los dispositivos punto a punto, vamos a especificar dos aspectos:

Attach(Ptr<PointToPointChannel> *canal***)**: es la función con la que conectamos el dispositivo de red con el canal que va a emplear.

Para el caso de estos dispositivos, tenemos dos protocolos a usar en los paquetes:

- 0x0800: representa a IPv4.
- 0x86DD: representa IPv6.

SetDataRate(DataRate *velocidad***)**: permite configurar la velocidad de transmisión a la que enviará paquetes dicho dispositivo.

SetQueue(Ptr<Queue<Packet>> *cola***)**: añade una cola de recepción al elemento punto a punto.

GetQueue(): para obtener una copia de la cola asociada al dispositivo.

Receive(Ptr<Packet> *paquete*): recibe un paquete del canal conectado. Este método es utilizado por el propio canal para indicar que el último bit de un paquete ha llegado al dispositivo.

SetReceiveCallback(MakeCallback(bool *función***))**: cada vez que le llega un paquete y, por tanto, se deba entregar a capas superiores mediante *MakeCallback*, llamamos a la función que debe procesar el paquete recibido. Podemos destacar que la función debe devolver el booleano *true* para su buen funcionamiento. Si dicho método necesita de alguna variable como argumento, se proporcionará en los siguientes campos de *MakeCallback*.

4.3 Application

Es la clase a utilizar para las aplicaciones que diseñemos. Sus funciones son:

GetNode(): devuelve el nodo al que está asociado dicha aplicación.

SetNode(Ptr<Node> *nodo***)**: se asocia el nodo a la aplicación correspondiente. Esta función es referenciada desde *Node::AddApplication*, para vincular el nodo y la aplicación.

SetStartTime(Time *tiempo***)**: tiempo en el que empezará a funcionar la aplicación.

SetStopTime(Time *tiempo***)**: tiempo en el que cesará la actividad de la aplicación.

Dentro de la aplicación, podemos acceder al nodo que la contiene con la variable *m_node*, que actuará de igual manera que un dispositivo de la clase nodo.

Hay que destacar dos funciones privadas, que son fundamentales para el desarrollo de una aplicación:

StartApplication(): es el primer método al que llama la aplicación cuando se ejecuta. Lo habitual es que llame a otra función que forme parte del comportamiento habitual de la aplicación.

StopApplication(): cuando termina el tiempo de ejecución de la aplicación llama a esta función antes de finalizar.

Cuando diseñemos una aplicación es muy importante definir correctamente dichas funciones para el buen funcionamiento de esta.

Las aplicaciones que vienen definidas en el software son las siguientes:

Tabla 4-1. Aplicaciones del software ns3

Nombre	Aplicación
EpsBearerTagUdpClient	Cliente UDP[2], incluyendo etiquetas para el uso con LTE (*Long Term Evolution*).
BsmApplication	Envía y recibe mensajes de seguridad básicos (*BSM: Basic Security Messages*) del IEEE 1609 WAVE[3].
BulkSendApplication	Manda todo el tráfico posible, intentando llenar el ancho de banda.
EpcEnbApplication	Aplicación instalada en los eNB (*evolved Node B)* con el fin dotarlos de funcionalidad de puente para los paquetes del plano de usuario entre la interfaz radio y la interfaz S1-U.
EpcSgwPgwApplication	Implementa la funcionalidad SGW/PGW (*Serving Gateways/Packet Data Network Gateway*), propio de la tecnología LTE, al igual que la aplicación anterior.
OnOffApplication	Genera tráfico a un único destino de acuerdo a un patrón On/Off.
PacketSink	Recibe y consume tráfico generado a una dirección IP y un puerto.

[2] *UDP: User Datagram Protocol*, protocolo básico de nivel de transporte.
[3] *WAVE: Wireless Access in Vehicular Environments*, estandarización de un grupo de protocolo de acceso inalámbrico en entornos vehiculares.

Tabla 4-1. Aplicaciones del *software* ns3 (continuación)

PacketSocketClient	Cliente**.**
PacketSocketServer	Servidor.
Ping6	Aplicación Ping en IPv6.
Radvd	*Router advertisement Daemon* implementa anuncios de enlace local de direcciones de enrutador IPv6 y prefijos de enrutamiento IPv6, utilizando el Protocolo de detección de vecinos (NDP: *Neighbor Discovery Protocol*).
SocketWriter	Para escribir en sockets TCP.
UdpClient	Cliente UDP.
UdpEchoClient	Cliente UDP de tipo Echo.
UdpServer	Servidor UDP.
UdpEchoServer	Servidor UDP de tipo Echo.
UdpTraceClient	Servidor UDP con una traza para poder rastrear.
V4Ping	Una aplicación que envía una solicitud ICMP ECHO, espera la respuesta y calcula el tiempo de ida y vuelta.
Receiver	Receptor de tipo Wi-Fi.
Sender	Transmisor de tipo Wi-Fi.

Además de todas estas aplicaciones que vienen por defecto, podemos diseñar nuestra propia aplicación, que constará de los atributos y funciones de la clase *Application.*

4.4 Channel

Este es un canal de comunicaciones ideal, que no introduce errores. Al igual que los dispositivos de red, el canal variará según el uso que le demos para cada comunicación. Por tanto, tendremos *PointToPointChannel, WimaxChannel, CsmaChannel…*Puede representar un canal tan simple como un cable, o tan complejo como el espacio-tiempo.

Podemos encontrar una serie de métodos generales para todos los canales:

GetId(): devuelve el identificador único de este canal.

GetNDevice(): proporciona el número de dispositivos asociados al canal.

Además de dicho canal, tenemos otras dos clases:

ErrorChannel: introduce un retraso determinista en los paquetes pares/impares.

SimpleChannel: es un modelo simplificado de un canal, que solo puede utilizarse con dispositivos del mismo

tipo, *SimpleNetDevice*, caracterizados por tener direcciones MAC[4] de 48 bits.

4.4.1 PointToPoint Channel

Si particularizamos para una comunicación punto a punto, cabe destacar el atributo *Delay*, que representa el retardo que sufre un paquete cuando atraviesa un canal. Podemos alterar su valor de la siguiente forma:

SetAttribute("Delay", TimeValue(Time *retardo***));**

Attach(Ptr<NetDevice> *dispositivo***)**: es la función con la que conectamos el dispositivo de red con el canal que va a emplear. Podemos usar indistintamente este método o *NetDevice::Attach*.

GetDelay(): devuelve el retardo asociado a dicho canal.

4.4.2 ErrorModel

Los canales especificados solo pueden introducir errores de retraso en los paquetes. Para tener paquetes erróneos, necesitamos la clase *ErrorModel*, la cual introduce una bandera en los paquetes perdidos o erróneos.

Su método principal es:

IsCorrupt(Ptr<Packet> *paquete***)**: devuelve el booleano *true* si, teniendo en cuenta el modelo empleado, considera que el paquete está corrupto.

Otras funciones útiles de esta clase son:

Disable(): deshabilita el modelo de error.

Enable(): habilita el modelo de error asociado.

IsEnabled(): devuelve *true* si el modelo está habilitado.

Tenemos cuatro tipos de modelos de error más específicos: *RateErrorModel*, *BurstErrorModel*, *ListErrorModel* y *ReceiveListErrorModel*. Si queremos asociarle una tasa de error a elección del usuario, utilizaremos la subclase *RateErrorModel*. Gracias a ella, podemos establecer la tasa de error, la unidad o una distribución (*RandomVariableStream*).

Además de las funciones heredadas, esta clase hija tiene sus métodos propios:

SetRandomVariable(Ptr<RandomVariableStream> *variable***)**: se decidirá si los paquetes son erróneos o no según la distribución aleatoria. Por defecto, es una variable uniforme (0,1).

SetRate(double *tasa***)**: los paquetes serán corruptos siguiendo la tasa que le asociamos.

SetUnit(ErrorUnit *unidad***)**: establecemos la unidad en la que se decidirá si los paquetes son erróneos o no. Tiene tres valores posibles: ERROR_UNIT_BIT, ERROR_UNIT_BYTE, ERROR_UNIT_PACKET.

GetRate(): devuelve un double con la tasa agregada al modelo de error.

GetUnit(): nos indica sobre qué unidad estamos aplicando el modelo.

En la Figura 4-3 podemos ver un ejemplo de utilización de esta clase.

[4] *MAC: Media Access Control, identificador único de un dispositivo de red.*

```
#include <ns3/core-module.h>
#include <ns3/error-model.h>
#include <ns3/packet.h>

using namespace ns3;

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

    int i;
    double contador =0;
    double contadorerr =0;

    Ptr<RateErrorModel> error = CreateObject<RateErrorModel> ();
    error->SetUnit(RateErrorModel::ERROR_UNIT_PACKET);
    error->SetRate (0.05);

    if (!error->IsEnabled())
        std::cout << "El modelo de error está desactivado."
                  << std::endl;

    for(i=1; i<=1e6; i++)
    {
        Ptr<Packet> paquete = Create<Packet>();
        if (error->IsCorrupt(paquete))
            contadorerr++;
        contador++;
    }

    std::cout << "Tasa de error: " << contadorerr/contador*100
              << "%"<< std::endl;

return 0;
}
```

Figura 4-3: Código de ejemplo11.cc

4.5 Queue

La clase *Queue* determina el funcionamiento de las colas de almacenamiento de los dispositivos de red. El tipo de objetos que se pueden almacenar van especificados en su definición, pudiendo ser cualquier clase que tenga el método *GetSize*.

```
Ptr<Queue<Item>> nombreCola = CreateObject<Queue>();
```

En general, lo que el usuario desea capturar es el tráfico que pasa por un dispositivo, es decir, el flujo de paquetes. Para ello, en la definición, sustituiremos *Item* por la clase que queremos, *Packet*.

Dequeue(): elimina un ítem de la cola de almacenamiento.

DoDequeue(ConstIterator *posición***)**: desencola el elemento situado en la posición indicada.

Enqueue(Ptr<*Item***>** *ítem***)**: coloca en la cola el elemento que indiquemos.

DoEnqueue(ConstIterator *posición***, Ptr<***Item***>** *ítem***)**: encola en la posición especificada el objeto que deseemos.

Si asociamos la cola a un dispositivo de red con el método *SetQueue*, estas funciones se llamarán automáticamente para ir encolando y desencolando paquetes (suponiendo el caso habitual).

4.5.1 DropTailQueue

Es una de las clases que definen colas de almacenamiento de paquetes. Concretamente, es una cola de tipo FIFO (*First In First Out*), en la cual, si se produce desbordamiento, los paquetes que se desechan son los últimos que han llegado.

4.6 NetDeviceContainer

La clase *NetDeviceContainer* nos permite crear un grupo englobando a varios dispositivos de igual tipo y con las mismas características. Es especialmente relevante cuando se necesita definir un gran número de componentes en una red. Cada elemento del grupo será del tipo *Ptr<NetDevice>*.

4.6.1 NodeContainer

Es una particularidad de la anterior, que nos permite crear un conjunto de nodos iguales.

Add(NodeContainer *otroContenedor***)**: añade el contenido de otro conjunto de nodos al final del propio.

Add(Ptr<Node> *nodo***)**: adjunta otro nodo en el grupo de interés.

Begin(): devuelve un valor de tipo iterador, apuntando al primer elemento del contenedor.

End(): obtiene un iterador que indica el último elemento del grupo.

Create(uint32_t *número***)**: crea un contenedor con el número de nodos que queramos.

Get(uint32_t *índice***)**: obtiene el nodo ubicado en la posición *índice*.

GetN(): devuelve el número de nodos almacenados en el grupo indicado.

4.7 Helper

Las clases de este software pueden venir seguidas de la palabra *helper*. Estas constituirán unas clases diferentes, cuyo objetivo es facilitar la labor del programador, simplificando la configuración de ciertos atributos.

Entre ellas, podemos encontrar las clases *PointToPointHelper*, *CsmaHelper, WifiHelper*, *RoutingHelper*…

Todas hacen uso de *NetDeviceContainer*, o, en su defecto, *NodeContainer*, de manera que la combinación de ambas clases contribuye a favorecer el desarrollo de programas en ns3.

4.7.1 PointToPointHelper

En el caso concreto para una comunicación punto a punto, podemos destacar los siguientes métodos:

Install(NodeContainer *contenedor***)**: crea un canal punto a punto entre todos los nodos que configuran el grupo que le indiquemos.

SetChannelAttribute(String *nombre***, AttributeValue** *valor***)**: permite modificar el contenido por defecto de los atributos del canal.

SetDeviceAttribute(String *nombre***, AttributeValue** *valor***)**: accede a los atributos de los dispositivos para poder cambiar su valor asociado. Dichos atributos dependerán del tipo de dispositivo empleado en cada momento.

```
#include <ns3/core-module.h>
#include <ns3/point-to-point-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
    Time tiempo("1ms");
    DataRate velocidad ("3Mbps");

    NodeContainer nodos;
    nodos.Create(2);

    PointToPointHelper comunicacion;
    comunicacion.SetChannelAttribute("Delay",
                                     TimeValue(tiempo));
    comunicacion.SetDeviceAttribute("DataRate",
                                    DataRateValue(velocidad));

    comunicacion.Install(nodos);

return 0;
}
```

Figura 4-4: Código de ejemplo12.cc

5 SIMULACIÓN

One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.

Una máquina puede hacer el trabajo de cincuenta hombres corrientes. Pero no existe ninguna máquina que pueda hacer el trabajo de un hombre extraordinario.

- Elbert Hubbard -

Durante este capítulo, vamos a aplicar todos los conceptos desarrollados en los apartados anteriores para realizar una comunicación punto a punto entre una aplicación transmisora y una receptora. Estas aplicaciones no pertenecerán a las definidas por el software ns3, sino que serán diseñadas para esta simulación concreta. Consistirá en un transmisor que envía paquetes de forma continua, durante un determinado tiempo establecido. Para realizar las comparaciones entre unos parámetros y otros, se generará una gráfica de tipo gnuplot.

5.1 Aplicación transmisora

A la hora de diseñar una aplicación, los métodos más importantes son *StartApplication* y *StopApplication*, tal y como hemos mencionado previamente. Esto se debe a que ambas funciones definirán el comportamiento de la aplicación al inicio y fin de su ejecución.

En nuestro caso, el inicio marcará el primer paquete que el transmisor envía al canal, mientras que el final de la aplicación detendrá la simulación que lanzaremos en otro punto del programa.

```
void ClaseTransmisor::StartApplication()
{
    EnviarPaquete();
}

void ClaseTransmisor::StopApplication()
{
    Simulator::Stop();
}
```

Figura 5-1: Métodos fundamentales de *ClaseTransmisor*

Ambos métodos deben ser privados, por lo que solo podrán llamarse dentro de la ejecución del programa. Como ya hemos indicado en un apartado previo, estas funciones se llaman automáticamente cuando los tiempos de inicio y fin de ejecución se cumplen.

En el constructor de la clase, como es habitual, daremos los valores por defecto a las variables de dicha clase, a la que hemos llamado *ClaseTransmisor*. Estas variables las recogemos en la siguiente tabla:

Tabla 5-1. Variables de la aplicación transmisora

Variable	Funcionalidad
p2pTx	Dispositivo transmisor.
temporizador	Tiempo máximo de recepción de paquetes.

Tabla 5-1. Variables de la aplicación transmisora (continuación)

tamPaquetes	Tamaño del paquete en bytes.
asentimiento	Código numérico empleado para el ACK[5].
secuencia	Número de secuencia del paquete.
totalpaquetes	Número total de paquetes enviados.
totalACK	Número total de paquetes asentidos.
cuentaAtras	Temporizador de retransmisiones.

Por consiguiente, el constructor de *ClaseTransmisor* será de la siguiente forma:

```
ClaseTransmisor::ClaseTransmisor
    (Ptr<NetDevice> t_dispositivo, Time t_temporizador,
     int32_t t_tamPaquetes, uint8_t t_asentimiento)
{
    p2pTx = t_dispositivo;
    temporizador = t_temporizador;
    tamPaquetes = t_tamPaquetes;
    asentimiento = t_asentimiento;

    secuencia = SequenceNumber16();

    totalpaquetes = 0;
    totalACK = 0;
}
```

Figura 5-2: Constructor de *ClaseTransmisor*

El objetivo principal de un transmisor es mandar paquetes y recibir asentimientos. Ambos aspectos son los siguientes que vamos a tratar.

En el método *EnviarPaquete*, crearemos un paquete del tamaño que indique la variable correspondiente, y lo mandaremos por el elemento de red transmisor. A continuación, incrementaremos el número total de paquetes enviados, y programaremos el temporizador. La variable *cuentaAtras* es de tipo *EventId*, y la programaremos para que realice una cuenta atrás desde el valor de la variable *temporizador*. De esta manera, si el asentimiento no se ha recibido en el tiempo de retransmisiones que indica dicho temporizador, habrá que reenviar el paquete, pues supondrá que se ha perdido el anterior en el trayecto. Si, por el contrario, llega el ACK antes de que expire el temporizador, se cancelará en el método correspondiente.

Para distinguir unos paquetes de otros, añadiremos una cabecera del modo indicado en la sección pertinente (apartado 3.12.3). En dicha cabecera introduciremos un número de secuencia que irá aumentando conforme se envíen paquetes, es decir, los paquetes consecutivos tendrán números de secuencia consecutivos. El primer paquete enviado tendrá el número cero y, como hemos establecido que el número de secuencia sea de tipo uint_16, tras el paquete cuyo número es 65.535, el siguiente tendrá una cabecera con número de secuencia cero.

[5] ACK: *acknowledgement*, acuse de recibo o asentimiento.

```
void ClaseTransmisor::EnviarPaquete()
{
    Ptr<Packet> paquete = Create<Packet> (tamPaquetes);

    MyHeader cabecera = MyHeader();
    cabecera.SetData (secuencia.GetValue());
    paquete->AddHeader (cabecera);

    secuencia.operator++();

    p2pTx->Send(paquete, p2pTx->GetAddress(), 0x800);

    totalpaquetes++;

    if(temporizador!=0)
        cuentaAtras = Simulator::Schedule(temporizador,
                    &ClaseTransmisor::VenceTemporizador,this);
}
```

Figura 5-3: Método para enviar paquetes de *ClaseTransmisor*

Otra de las funciones del transmisor será el proceso de recibir un asentimiento. Un ACK es un mensaje que el receptor envía al transmisor para confirmar el recibo del paquete. Dicho mensaje es de común acuerdo entre los extremos de la comunicación, siendo siempre el mismo. En nuestro caso, en cada nueva comunicación, transmisor y receptor podrán establecer un nuevo código de acuse de recibo, pudiendo ser un número entre 0 y 255, pues hemos limitado su longitud a un único byte. En el método que estamos tratando, cuando el transmisor ha recibido un paquete, procede a copiar su contenido en un *buffer* y a compararlo con el código de asentimiento preestablecido. Si este coincide, se cancela el temporizador y se incrementa el contador de ACK recibidos, que nos indicará el número total de paquetes sin error que han llegado al receptor. Por último, el transmisor mandará otro paquete al receptor.

```
bool ClaseTransmisor::ACKRecibido
            (Ptr<NetDevice> receptor, Ptr <const Packet> paquete,
             uint16_t protocolo, const Address & desde)
{
    uint8_t contenido=0;

    paquete->CopyData(&contenido, 1);

    if (contenido == asentimiento)
    {
        Simulator::Cancel(cuentaAtras);
        totalACK++;
        EnviarPaquete();
    }

return true;
}
```

Figura 5-4: Método para recibir paquetes de *ClaseTransmisor*

Las dos últimas funciones de la aplicación transmisora son muy sencillas. La primera, *VenceTemporizador*, tan solo retransmite un paquete cuando el tiempo establecido por el temporizador ha expirado. Hay que tener en cuenta que, si retransmitimos un paquete, debe tener el mismo número de secuencia que el anterior, por lo que hay que modificar la variable que se encarga de gestionar dichos números.

Por otra parte, *paquetesUtiles* nos proporciona el número de paquetes útiles de la comunicación, cuyo cálculo se hará a partir del número de asentimientos recibidos y el número total de paquetes enviados.

```
void ClaseTransmisor::VenceTemporizador()
{
    NS_LOG_WARN("VenceTemporizador");
    secuencia.operator--();
    EnviarPaquete();
}


double ClaseTransmisor::paquetesUtiles ()
{
    NS_LOG_INFO("Porcentaje paquetes utiles: "
                << double(totalACK)/totalpaquetes*100 << "%");

return double(totalACK)/totalpaquetes*100;
}
```

Figura 5-5: Métodos de gestión de temporizador y paquetes de *ClaseTransmisor*

5.2 Aplicación receptora

El diseño del receptor es menos complejo que la aplicación anterior. Su comportamiento consiste en recibir paquetes y mandar asentimientos.

En el constructor de la clase, a la cual hemos nombrado como *ClaseReceptor*, se inicializarán las variables privadas de dicha clase. En la siguiente tabla se recogen estas variables y su finalidad:

Tabla 5-2. Variables de la aplicación receptora

Variable	Funcionalidad
p2pRx	Dispositivo receptor.
asentimiento	Código numérico empleado para el ACK.

```
ClaseReceptor::ClaseReceptor
    (Ptr<NetDevice> r_dispositivo, uint8_t r_asentimiento)
{
    p2pRx = r_dispositivo;
    asentimiento = r_asentimiento;
}
```

Figura 5-6: Constructor de *ClaseReceptor*

Como el funcionamiento del receptor es sencillo, lo hemos diseñado con un único método, que se encarga de recibir los paquetes enviados por el transmisor, crear el paquete de acuse de recibo con el código numérico preestablecido con el otro dispositivo y enviarlo. Como hemos indicado previamente, por conveniencia, hemos fijado la longitud del asentimiento en un byte.

```
bool ClaseReceptor::PaqueteRecibido
        (Ptr<NetDevice> receptor, Ptr <const Packet> paquete,
         uint16_t protocolo, const Address & desde)
{
    NS_LOG_DEBUG("Paquete Recibido");

    Ptr<Packet> paqueteAck = Create<Packet> (&asentimiento, 1);
    p2pRx->Send(paqueteAck, p2pRx->GetAddress(), 0x800);

return true;
}
```

Figura 5-7: Método para gestionar paquetes de *ClaseReceptor*

5.3 Escenario

Estas aplicaciones transmisora y receptora habrá que asociarlas a un escenario concreto. En nuestro caso, vamos a llevar a cabo una simulación de una comunicación punto a punto, por lo que habrá que definir un canal y dos dispositivos punto a punto, así como dos nodos y las aplicaciones que hemos explicado. Tras asociar los dispositivos y las aplicaciones a los nodos, vamos a agregar también colas de recepción a cada nodo.

```
Ptr<PointToPointChannel> canal = CreateObject<PointToPointChannel>();

Ptr<Node> nodoTx = CreateObject<Node>();
Ptr<Node> nodoRx = CreateObject<Node>();

Ptr<PointToPointNetDevice> p2pTx =
                            CreateObject<PointToPointNetDevice>();
Ptr<PointToPointNetDevice> p2pRx =
                            CreateObject<PointToPointNetDevice>();

Ptr<RateErrorModel> probError = CreateObject<RateErrorModel> ();
probError->SetUnit(RateErrorModel::ERROR_UNIT_PACKET);
probError->SetRate (e_error);
p2pTx->SetReceiveErrorModel(probError);

ClaseTransmisor appTransmisor(p2pTx, e_temporizador,
                              e_tampaquetes, e_asentimiento);
ClaseReceptor appReceptor(p2pRx, e_asentimiento);

nodoTx->AddDevice(p2pTx);
nodoRx->AddDevice(p2pRx);

nodoTx->AddApplication(&appTransmisor);
nodoRx->AddApplication(&appReceptor);

p2pTx->SetQueue(CreateObject<DropTailQueue>());
p2pRx->SetQueue(CreateObject<DropTailQueue>());
```

Figura 5-8: Escenario del programa.

Gracias a la función *SetReceiveCallback*, vamos a conseguir que la comunicación se mantenga continuamente. Estableceremos que, cada vez que el transmisor reciba un mensaje, llame a la función *ACKRecibido*, ya que los asentimientos son los únicos paquetes que le llegarán a dicho dispositivo. Por su parte, el receptor, cuando reciba un paquete, será de datos útiles, y tendrá que enviar el acuse de recibo correspondiente. Para ello, cada vez que reciba un paquete irá a la función *PaqueteRecibido*.

A continuación, modificaremos los valores por defecto del retraso del canal, así como la tasa de envío del dispositivo transmisor. Estableceremos el tiempo de inicio y fin de las aplicaciones, en el que se llamará a las funciones *StartApplication* y *StopApplication*. En el tiempo de inicio se llamará por primera vez al método que envía paquetes en el transmisor, y, gracias a las funciones de *SetReceiveCallback*, se realimentarán las aplicaciones una a otra hasta que se alcance el tiempo final. Puede llamar la atención que los métodos *StartApplication* y *StopApplication* no se hayan definido para la aplicación receptora. Esto se debe a que esta no realiza nada hasta que le llega un paquete, lo cual depende de la aplicación transmisora. Por esta causa, el receptor no necesita realizar ninguna operación al empezar la ejecución.

```
p2pTx->SetReceiveCallback(
      MakeCallback(&ClaseTransmisor::ACKRecibido, &appTransmisor));
p2pRx->SetReceiveCallback(
      MakeCallback(&ClaseReceptor::PaqueteRecibido, &appReceptor));

p2pTx->Attach(canal);
p2pRx->Attach(canal);

canal->SetAttribute("Delay", TimeValue(e_retardo));
p2pTx->SetDataRate(e_velocidad);

appTransmisor.SetStartTime(Seconds(1));
appTransmisor.SetStopTime(Seconds(10));

Simulator::Run();

Simulator::Destroy();
```

Figura 5-9: Configuración de parámetros y programación de la simulación

Puesto que nuestro objetivo es comparar el funcionamiento del sistema para distintos valores del temporizador de retransmisiones y velocidad de transmisión, dicho escenario formará parte de una función externa a la principal, a la que se llamará con distintos valores para estos parámetros. Para poder realizar la comparación, utilizaremos el porcentaje de paquetes útiles obtenido con cada uno de los valores.

```
double escenario (double e_velocidad, Time e_retardo,
                  Time e_temporizador, uint32_t e_tampaquetes,
                  uint8_t e_asentimiento, double e_error)
{

...

return appTransmisor.paquetesUtiles();

}
```

Figura 5-10: Formato del escenario

5.4 Main

En la función principal, nos centraremos en crear los bucles necesarios para llamar al escenario con distintos valores para los parámetros según los que queremos hacer la comparación. Con este fin, podremos introducir valores por línea de comando o dejar los que están establecidos por defecto. A partir de los valores de dichos parámetros, calcularemos el intervalo de saltos necesario para el número de iteraciones que hayamos establecido.

Detallaremos las variables que se utilizarán en este apartado en las siguientes tablas.

Tabla 5-3: Variables del programa principal

Variable	Funcionalidad
velocidadInicial	Velocidad de transmisión inicial.
velocidadFinal	Velocidad de transmisión final.
retardo	Retardo del canal de propagación.
temporizadorInicial	Temporizador de retransmisión de paquetes inicial.
temporizadorFinal	Temporizador de retransmisión de paquetes final.
numvelocidad	Número de saltos entre velocidades inicial y final.
numtiempo	Número de saltos entre el temporizador inicial y final.
error	Probabilidad de error del paquete.
asentimiento	Código para el acuse de recibo.
tamPaquetes	Tamaño de los paquetes transmitidos.

Tabla 5-4: Variables auxiliares del programa principal

Variable	Funcionalidad
cmd	Identificador de la línea de comandos.
grafica	Gráfica gnuplot que incluye las distintas curvas.
nombrecurva	Cadena para nombrar una curva.
curva	Curva de una velocidad de transmisión dada.
fichero	Fichero gnuplot.
titulo	Título de la gráfica.
asentauxiliar	Código de asentimiento auxiliar para realizar comprobaciones.
saltovelocidad	Intervalo entre dos velocidades distintas.
saltotiempo	Intervalo entre dos temporizadores distintos.
paquetesUtiles	Porcentaje de paquetes útiles para unos parámetros dados.
i	Variable auxiliar para la iteración de velocidades.
j	Variable auxiliar para la iteración del temporizador.

En el programa se da la opción de cambiar los valores por defecto de las variables implicadas en el cálculo de paquetes útiles. Estas son las que se detallan en la Figura 5-11.

```
CommandLine cmd;

cmd.AddValue("velocidadInicial",
             "Velocidad de transmisión inicial", velocidadInicial);
cmd.AddValue("velocidadFinal",
             "Velocidad de transmisión final", velocidadFinal);
cmd.AddValue("retardo", "Retardo de propagación", retardo);
cmd.AddValue("temporizadorInicial",
             "Tiempo de retransmisión inicial", temporizadorInicial);
cmd.AddValue("temporizadorFinal",
             "Tiempo de retransmisión final", temporizadorFinal);
cmd.AddValue("numvelocidad",
             "Número de saltos de velocidad", numvelocidad);
cmd.AddValue("numtiempo",
             "Número de saltos de temporizador de retransmisiones",
             numtiempo);
cmd.AddValue("error", "Probabilidad de error de paquetes", error);
cmd.AddValue("asentauxiliar",
             "Código empleado como acuse de recibo", asentauxiliar);
cmd.AddValue("tamPaquetes",
             "Tamaño de los paquetes enviados", tamPaquetes);
cmd.Parse(argc, argv);
```

Figura 5-11: Paso de parámetros por línea de comandos

Si se añaden valores a dichos parámetros, debemos comprobar que los números introducidos son válidos. Esto se realiza con la siguiente consecución de bucles *if-else*, que comprueban varios casos, tales como que los valores finales sean mayores que los iniciales y que estos sean mayores que cero, entre otros.

```
if (velocidadInicial.GetBitRate() == 0)
    NS_LOG_WARN("La velocidad inicial debe ser mayor que cero");
else if(velocidadInicial.GetBitRate() > velocidadFinal.GetBitRate())
    NS_LOG_WARN("La velocidad final debe ser mayor que la inicial");
else if(temporizadorInicial.GetDouble() == 0)
    NS_LOG_WARN("El temporizador inicial debe ser mayor que cero");
else if (temporizadorInicial.GetDouble() > temporizadorFinal.GetDouble())
    NS_LOG_WARN("El temporizador final debe ser mayor que el inicial");
else if (numvelocidad < 0)
    NS_LOG_WARN("Los saltos de velocidad no pueden ser negativos");
else if ((numvelocidad == 0) &&
         (velocidadInicial.GetBitRate() != velocidadFinal.GetBitRate()))
    NS_LOG_WARN("Entre dos velocidades distintas debe haber al menos un salto");
else if (numtiempo <= 0)
    NS_LOG_WARN("Los saltos de tiempo deben ser mayor que cero");
else if ((error < 0) || (error >= 1))
    NS_LOG_WARN("La probabilidad de error debe estar en el intervalo [0,1)");
else if ((asentauxiliar < 0) || (asentauxiliar > 255))
    NS_LOG_WARN("El código de ACK debe estar en el intervalo [0,255]");
else if (tamPaquetes <= 0)
    NS_LOG_WARN("El tamaño de los paquetes debe ser mayor que cero");
```

Figura 5-12: Gestión de los parámetros erróneos

Si estas condiciones se cumplen, el programa pasará a calcular los intervalos de velocidad y tiempo necesarios con las condiciones inicial y final, así como el número de saltos entre estas. Si, por el contrario, no se cumple alguna, se gestionará el aviso correspondiente a mostrar por los registros, y se finaliza la ejecución.

Los saltos entre un temporizador de retransmisión y otro serán equiespaciados, mientras que, para el caso de la velocidad, debemos tener en cuenta el caso en el que solo consideremos una curva, es decir, solo una

velocidad. En este último, los saltos serán cero, pero debemos considerarlo un salto mínimo para poder salir del bucle.

```
asentimiento = asentauxiliar;

if(numvelocidad!=0)
    saltovelocidad=((double)velocidadFinal.GetBitRate()-
        (double)velocidadInicial.GetBitRate())/numvelocidad;
else
    saltovelocidad=1;

saltotiempo = (temporizadorFinal.GetDouble()-
        temporizadorInicial.GetDouble())/numtiempo;


Gnuplot grafica ("TrabajoFindeGrado.png");
```

Figura 5-13: Preparación de la iteración

Gracias a dichos saltos, podemos incluir dos estructuras de tipo *for*, para ir recorriendo los valores del intervalo que hayamos introducido. En cada una de las iteraciones, calcularemos el número de paquetes útiles para los valores dados, llamando, en cada caso, al escenario, y llevando a cabo la comunicación entre transmisor y receptor.

Cada uno de estos puntos los añadiremos a una gráfica para que el resultado se pueda analizar de forma visual y sencilla. A cada una de las curvas, que representará una velocidad de transmisión distinta, le asociaremos un nombre u otro, dependiendo del orden de magnitud de cada una, distinguiendo, por tanto, entre bps, kbps, Mbps y Gbps.

```
for (i=(double)velocidadInicial.GetBitRate();
     i<=(double)velocidadFinal.GetBitRate(); i=i+saltovelocidad)
{
    std::ostringstream nombrecurva;
    Gnuplot2dDataset curva;
    for (j = temporizadorInicial.GetDouble();
         j<= temporizadorFinal.GetDouble();
         j=j+saltotiempo)
    {
        paquetesUtiles = escenario
            (i, retardo, Time(j), tamPaquetes, asentimiento, error);
        curva.Add(j, paquetesUtiles);
    }
    grafica.AddDataset(curva);

    if (i<1e3)
        nombrecurva << i << "bps";
    else if (i<1e6)
        nombrecurva << i/1e3 << "kbps";
    else if (i<1e9)
        nombrecurva << i/1e6 << "Mbps";
    else
        nombrecurva << i/1e9 << "Gbps";

    curva.SetTitle(nombrecurva.str());
```

Figura 5-14: Gestión de la iteración con distintos parámetros

Por último, prepararemos el fichero que contendrá la gráfica generada, añadiendo, de la siguiente manera, la leyenda, los ejes, el título y el rango entre los que se debe representar la variable *y*. Esto último se realiza con el fin de tener la misma escala vertical (0-100%) para todas las gráficas.

```
std::ofstream fichero ("TrabajoFindeGrado.plt");
grafica.SetLegend ("Temporizador de retransmisiones (ms)",
                   "Porcentaje paquetes útiles (%)");
titulo << "Retardo del canal: " << retardo;
grafica.SetTitle(titulo.str());
grafica.SetExtra("set key outside");
grafica.AppendExtra("set yrange [0:100]");
grafica.GenerateOutput(fichero);
fichero.close();
```

Figura 5-15: Generar e imprimir la gráfica

De esta manera, si ejecutamos la simulación con los valores por defecto, mostrados en la Tabla 5-5, obtendremos la gráfica presentada en la siguiente figura.

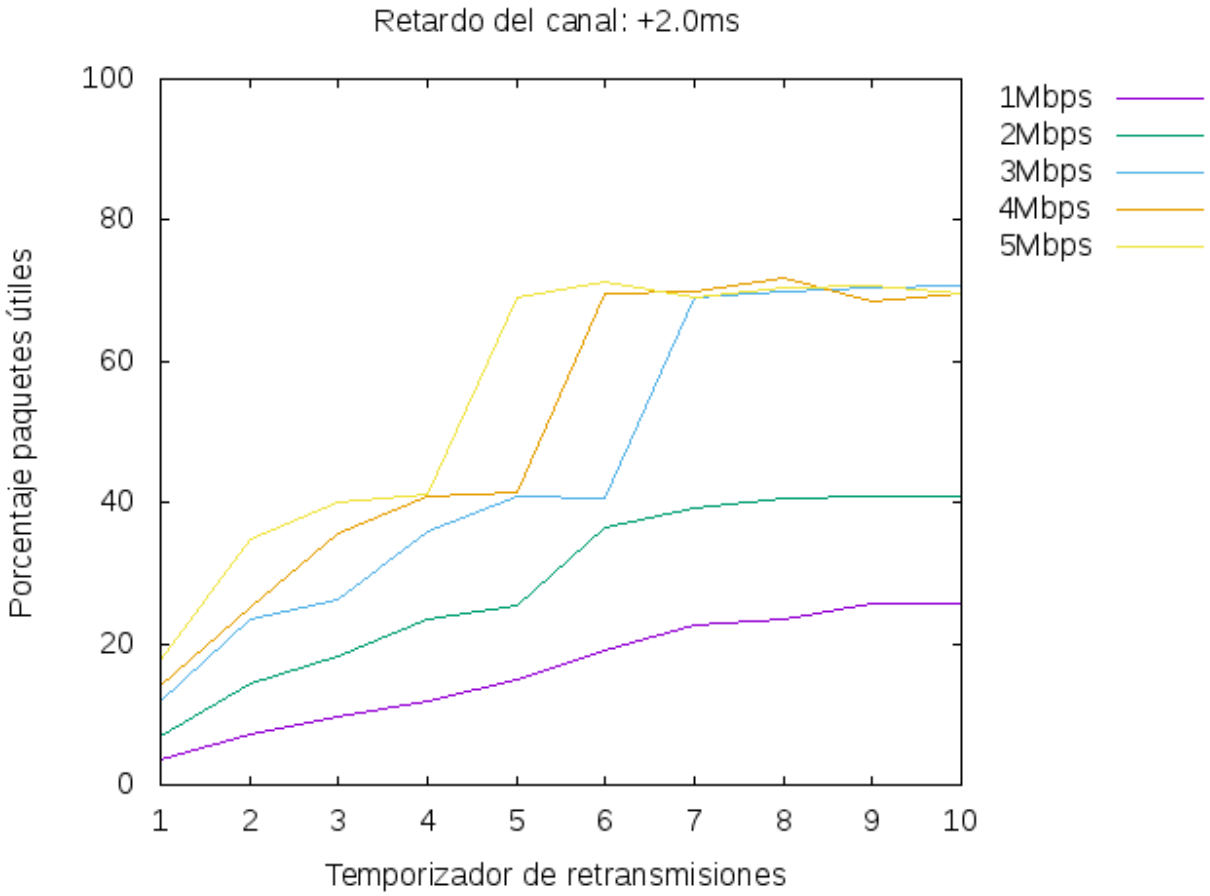

Figura 5-16: Gráfica de porcentaje de paquetes útiles frente al temporizador de retransmisiones

Tabla 5-5. Valores por defecto del programa

Variable	**Valor**	**Variable**	**Valor**
velocidadInicial	1Mbps	velocidadFinal	5Mbps
temporizadorInicial	1ms	temporizadorFinal	10ms
numvelocidad	4	numtiempo	9
retardo	2ms	error	0.3
asentimiento	110	tamPaquetes	2500

6 CONCLUSIÓN

Quality is never an accident. It is always the result of intelligent effort.

La calidad nunca es un accidente, siempre es el resultado de un esfuerzo de la inteligencia.

- John Ruskin -

El uso del software ns3 nos permite realizar simulaciones realistas, lo que favorece su utilización en ambientes educativos y de investigación. El diseño realizado a lo largo de este documento ha consistido en un transmisor, cuyo objetivo es enviar paquetes con números de secuencia consecutivos, y en un receptor, que manda acuses de recibo para los paquetes recibidos.

Como podemos comprobar en la última figura del apartado anterior, el porcentaje de paquetes correctos será mayor cuando aumente la velocidad de transmisión, así como cuando se incremente el temporizador de retransmisiones. Esto se debe a que, al aumentar dichos parámetros, hay más probabilidades de que el paquete llegue al destino antes de que expire el temporizador, ya sea porque este es más largo o porque la transmisión se realiza de forma más rápida.

Para otras posibles líneas de trabajo de este documento, cabe destacar que la utilización de ns3 abarca numerosos y extensos campos. Por lo tanto, como evolución para el futuro, se proseguirá aprendiendo y desarrollando programas en dicho software, con el fin de acercar el uso de este a los actuales estudiantes de la universidad.

Como mejoras factibles del proyecto realizado, podemos ahondar en el estudio de las redes punto a punto, para obtener un modelo más realista. Este objetivo se logrará a través de la consideración de todo tipo de errores debidos a la transmisión. Además, otra posibilidad sería el desarrollo de una simulación de tipo *full-dúplex*, es decir, una comunicación donde ambos extremos son, simultáneamente, transmisor y receptor.

Podemos mencionar que, en el futuro, este campo de trabajo se podría ampliar, por ejemplo, al estudio de una comunicación de tipo Wi-Fi[6], al empleo de una comunicación tipo cliente-servidor mediante el uso de *sockets* o al uso de un canal CSMA/CD[7]. Un aspecto que habría que destacar es la utilización de trazas para monitorizar ciertos aspectos de la comunicación, tales como tirar un paquete, generar un paquete…

Durante la realización de este trabajo, se han llevado a la práctica numerosos aspectos tratados de forma teórica durante la carrera. Desde un punto de vista personal, este hecho ha contribuido a asentar y ampliar los conocimientos que he adquirido a lo largo de estos años.

[6] Wi-Fi: *Wireless Fidelity* es un tipo de comunicación inalámbrica de área local.

[7] CSMA/CD*: Carrier Sense Multiple Access with Collision Detection,* algoritmo de acceso al medio compartido en el que los dispositivos de red escuchan el medio antes de transmitir para determinar si el canal se encuentra libre. Se emplea en tecnologías como Ethernet.

7 REFERENCIAS

[1] «Nsnam», [En línea]. Available: http://www.nsnam.org/

[2] «Documentation», [En línea]. Available: http://www.nsnam.org/documentation/

[3] «Doxygen», [En línea], Available: https://www.nsnam.org/docs/release/3.26/doxygen/

[4] «Wiki», [En línea]. Available: http://www.nsnam.org/wiki/

[5] «Code», [En línea]. Available: http://code.nsnam.org/

[6] «Manual», [En línea]. Available: https://www.nsnam.org/docs/manual/html/

[7] «Tutorial», [En línea]. Available: https://www.nsnam.org/docs/release/3.26/tutorial/html/index.html

[8] «Models», [En línea]. Available: https://www.nsnam.org/docs/release/3.26/models/html/index.html

[9] F.J. Ceballos, «Programación Orientada a Objetos con C++», Ra-Ma, 4ª edición, 2007

[10] L. Joyanes, I. Zahonero, «Estructura de datos en C++», McGraw-Hill, 1ª edicion, 2007

[11] J. Liberty, «Programación C++», Anaya Multimedia, 2011

[12] G. Medinabeitia, «Apuntes de clase, Fundamentos de Programación I», Universidad de Sevilla, curso 2013-2014

[13] F.J. Payán, «Principios de comunicaciones digitales. I, Fundamentos», Secretariado de Publicaciones de la Universidad de Sevilla, 2014

[14] C.Pérez, J.M. Zamanillo, A. Casanueva, «Sistemas de telecomunicación», Servicio de Publicaciones de la Universidad de Cantabria, 2007

[15] J.G. Proakis, «Digital communications», McGraw-Hill, 4ª edición, 2001

[16] F.J. Payán, «Apuntes de clase, Comunicaciones Digitales», Universidad de Sevilla, curso 2015-2016.

[17] A. Artés, F. Pérez, «Comunicaciones digitales», Pearson-Prentice Hall, 2007

[18] D. Ríos, «Simulación: métodos y aplicaciones», Ra-Ma, 2ª edición, 2008

[19] G. Kizer, «Digital Microwave Communication. Engineering Point-to-Point Microwave System», Wiley, 2013

[20] M. E. Moreno, «Apuntes de clase, Estadística», Universidad de Sevilla, curso 2013-2014

[21] A. Lapidoth, «A Foundation In Digital Communication», Cambridge University Press, 2009

[22] R. G. Gallager, «Principles of Digital Communications», Cambridge, 2008

[23] H. Nguyen, E. Shwedyk, «A First Course In Digital Communications», Cambridge University Press, 2009

[24] A. M. Law, «Simulation Modeling and Analysis», McGraw-Hill, 5ª edición, 2015

[25] A. Papoulis, «Probability, Random Variables and Stochastic Process», McGraw-Hill, 4ª edición, 2002

[26] R. E. Shannon, «Introduction to the Art and Science of Simulation», Proceedings of the 1998 Winter Simulation Conference, 1998

[27] S. Haykin, «Communication Systems», John Wiley and Sons, 4ª edición, 2000

Anexo A: Código

```
/*****************************************************************************
**
**          Fichero: trabajo.cc
**
**          Autor: Belen Rodriguez Estevez
**
**          Descripcion: Escenario y puesta en marcha de una comunicacion
**                       entre un transmisor, que envia paquetes
**                       de forma ininterrumpida; y un receptor, que manda
**                       ACK para que el transmisor no reenvie el paquete
**                       anterior.
**
**
*****************************************************************************/

/* Incluimos el fichero de cabecera */
#include "auxiliar.h"

/* Definimos el nombre para los registros de este fichero */
NS_LOG_COMPONENT_DEFINE("TrabajoFindeGrado");


/* Funcion principal */
int main(int argc, char *argv[])
{
        /* Ajustamos la resolucion del reloj a milisegundos */
        Time::SetResolution(Time::MS);

        /* Variables con valores por defecto*/
        DataRate velocidadInicial("1Mbps");
        DataRate velocidadFinal("5Mbps");
        Time retardo("2ms");
        Time temporizadorInicial("1ms");
        Time temporizadorFinal("10ms");
        int numvelocidad = 4;
        int numtiempo = 9;
        double error = 0.3;
        int asentauxiliar = 110;
        uint8_t asentimiento = 110;
        uint32_t tamPaquetes = 2500;
        double saltovelocidad;
        double saltotiempo;
        double paquetesUtiles = 0;
        double i;
        double j;
        std::ostringstream titulo;
```

```
/* Si se introducen valores por linea de comandos */
CommandLine cmd;

cmd.AddValue("velocidadInicial", "Velocidad de transmision inicial", velocidadInicial);
cmd.AddValue("velocidadFinal", "Velocidad de transmision final", velocidadFinal);
cmd.AddValue("retardo", "Retardo de propagacion: ", retardo);
cmd.AddValue("temporizadorInicial", "Tiempo de retransmision inicial", temporizadorInicial);
cmd.AddValue("temporizadorFinal", "Tiempo de retransmision final", temporizadorFinal);
cmd.AddValue("numvelocidad", "Numero de saltos de velocidad", numvelocidad);
cmd.AddValue("numtiempo", "Numero de saltos de temporizador de retransmisiones", numtiempo);
cmd.AddValue("error", "Probabilidad de error de paquetes", error);
cmd.AddValue("asentauxiliar", "Codigo empleado como acuse de recibo", asentauxiliar);
cmd.AddValue("tamPaquetes", "Tamano de los paquetes enviados", tamPaquetes);
cmd.Parse(argc, argv);

/* Comprobamos que los valores introducimos por linea de comandos son validos */
if (velocidadInicial.GetBitRate() == 0)
        NS_LOG_WARN("La velocidad inicial debe ser mayor que cero");
else if(velocidadInicial.GetBitRate() > velocidadFinal.GetBitRate())
        NS_LOG_WARN("La velocidad final debe ser mayor que la inicial");
else if(temporizadorInicial.GetDouble() == 0)
        NS_LOG_WARN("El temporizador inicial debe ser mayor que cero");
else if (temporizadorInicial.GetDouble() > temporizadorFinal.GetDouble())
        NS_LOG_WARN("El temporizador final debe ser mayor que el inicial");
else if (numvelocidad < 0)
        NS_LOG_WARN("Los saltos de velocidad no pueden ser negativos");
else if ((numvelocidad == 0) && (velocidadInicial.GetBitRate() != velocidadFinal.GetBitRate()))
        NS_LOG_WARN("Entre dos velocidades distintas debe haber al menos un salto");
else if ((numvelocidad > 0) && (velocidadInicial.GetBitRate() == velocidadFinal.GetBitRate()))
        NS_LOG_WARN("Entre dos velocidades iguales no puede haber saltos");
else if (numtiempo <= 0)
        NS_LOG_WARN("Los saltos de tiempo deben ser mayor que cero");
else if ((error < 0) || (error >= 1))
        NS_LOG_WARN("La probabilidad de error debe estar en el intervalo [0,1)");
else if ((asentauxiliar < 0) || (asentauxiliar > 255))
        NS_LOG_WARN("El codigo de ACK debe estar en el intervalo [0,255]");
else if (tamPaquetes <= 0)
        NS_LOG_WARN("El tamano de los paquetes debe ser mayor que cero");
else
{
        NS_LOG_INFO("Valores de las variables:");
        NS_LOG_INFO("Velocidad de transmision inicial: " << velocidadInicial.GetBitRate() << "bps");
        NS_LOG_INFO("Velocidad de transmision final: " << velocidadFinal.GetBitRate() << "bps");
        NS_LOG_INFO("Retardo de propagacion: " << retardo);
        NS_LOG_INFO("Tiempo de retransmision inicial: " << temporizadorInicial.GetDouble() << "ms");
        NS_LOG_INFO("Tiempo de retransmision final: " << temporizadorFinal.GetDouble() << "ms");
        NS_LOG_INFO("Numero de saltos de velocidad: " << numvelocidad);
        NS_LOG_INFO("Numero de saltos de temporizador de retransmisiones: " << numtiempo);
        NS_LOG_INFO("Probabilidad de error de paquetes: " << error*100 << "%");
        NS_LOG_INFO("Codigo empleado como acuse de recibo: " << asentauxiliar);
        NS_LOG_INFO("Tamano de los paquetes enviados: " << tamPaquetes);

        /* Asignamos el valor del acuse de recibo a traves de la variable auxiliar */
        asentimiento = asentauxiliar;

        /* Si hay saltos, calculamos el incremento entre salto y salto.
          Si las velocidades son iguales, establecemos un salto minimo */
```

```
        if(numvelocidad!=0)
                saltovelocidad=
                ((double)velocidadFinal.GetBitRate()-(double)velocidadInicial.GetBitRate())/numvelocidad;
        else
                saltovelocidad=1;

        /* Calculamos el incremento entre un tiempo y otro */
        saltotiempo = (temporizadorFinal.GetDouble()-temporizadorInicial.GetDouble())/numtiempo;

        NS_LOG_DEBUG("Incrementos velocidad: " << saltovelocidad);
        NS_LOG_DEBUG("Incrementos temporizador: " << saltotiempo);

        /* Creamos la grafica */
        Gnuplot grafica ("TrabajoFindeGrado.png");

        /* Iteramos para distintos valores de velocidad comprendidos entre la inicial y la final */
        for (i=(double)velocidadInicial.GetBitRate(); i<=(double)velocidadFinal.GetBitRate();
                i=i+saltovelocidad)
        {
                /* Creamos una curva para esta velocidad */
                std::ostringstream nombrecurva;
                Gnuplot2dDataset curva;

                /* Iteramos para distintos valores de temporizador comprendidos
                  entre el inicial y el final */
                for (j = temporizadorInicial.GetDouble(); j<= temporizadorFinal.GetDouble();
                        j=j+saltotiempo)
                {
                        /* Simulamos el escenario y la transmision con estos parametros*/
                        paquetesUtiles = escenario(i, retardo, Time(j), tamPaquetes, asentimiento, error);

                        /* Anadimos a la curva el numero de paquetes utiles con esta configuracion */
                        curva.Add(j, paquetesUtiles);

                        NS_LOG_DEBUG("Paquetes Utiles: " << paquetesUtiles);
                }

                /* Anadimos la curva a la grafica */
                grafica.AddDataset(curva);

                /* Segun el orden de magnitud de la velocidad, le ponemos nombre a la curva */
                if (i<1e3)
                        nombrecurva << i << "bps";
                else if (i<1e6)
                        nombrecurva << i/1e3 << "kbps";
                else if (i<1e9)
                        nombrecurva << i/1e6 << "Mbps";
                else
                        nombrecurva << i/1e9 << "Gbps";

                curva.SetTitle(nombrecurva.str());

        }

        /* Imprimimos la grafica */
std::ofstream fichero ("TrabajoFindeGrado.plt");
```

```
            /* Nombramos los ejes y a la propia grafica */
            grafica.SetLegend ("Temporizador de retransmisiones (ms)",
                                                    "Porcentaje paquetes utiles (%)");
            titulo << "Retardo del canal: " << retardo;
            grafica.SetTitle(titulo.str());

            /* Establecemos el rango del eje Y y colocamos la leyenda fuera del marco */
            grafica.SetExtra("set key outside");
            grafica.AppendExtra("set yrange [0:100]");

            grafica.GenerateOutput(fichero);
            fichero.close();
            }

return 0;
}


/* Funcion para crear el escenario y dar comienzo a la simulacion */
double escenario (double e_velocidad, Time e_retardo, Time e_temporizador, uint32_t e_tampaquetes,
                                        uint8_t e_asentimiento, double e_error)
{

            NS_LOG_FUNCTION(e_velocidad << e_retardo << e_temporizador << e_tampaquetes
                                                << e_asentimiento << e_error);

            /* Creamos el canal de comunicaciones */
            Ptr<PointToPointChannel> canal = CreateObject<PointToPointChannel>();

            /* Creamos los nodos transmisor y receptor */
            Ptr<Node> nodoTx = CreateObject<Node>();
            Ptr<Node> nodoRx = CreateObject<Node>();

            /* Creamos los dispositivos punto a punto */
            Ptr<PointToPointNetDevice> p2pTx = CreateObject<PointToPointNetDevice>();
            Ptr<PointToPointNetDevice> p2pRx = CreateObject<PointToPointNetDevice>();

            /* Creamos el modelo de error para un canal con perdidas */
            Ptr<RateErrorModel> probError = CreateObject<RateErrorModel> ();
            probError->SetUnit(RateErrorModel::ERROR_UNIT_PACKET);
            probError->SetRate (e_error);
            p2pTx->SetReceiveErrorModel(probError);

            /* Definimos las aplicaciones transmisor y receptor */
            ClaseTransmisor appTransmisor(p2pTx, e_temporizador, e_tampaquetes, e_asentimiento);
            ClaseReceptor appReceptor(p2pRx, e_asentimiento);

            /* Asociamos el dispositivo punto a punto con su nodo correspondiente */
            nodoTx->AddDevice(p2pTx);
            nodoRx->AddDevice(p2pRx);

            /* Asociamos el nodo con su aplicacion */
            nodoTx->AddApplication(&appTransmisor);
            nodoRx->AddApplication(&appReceptor);

            /* Anadimos las colas de recepcion */
            p2pTx->SetQueue(CreateObject<DropTailQueue>());
            p2pRx->SetQueue(CreateObject<DropTailQueue>());
```

```
        /* Siempre que reciben un paquete y hay que entregarlo a las capas superiores,
          llama a las funciones establecidas */
        p2pTx->SetReceiveCallback(MakeCallback(&ClaseTransmisor::ACKRecibido, &appTransmisor));
        p2pRx->SetReceiveCallback(MakeCallback(&ClaseReceptor::PaqueteRecibido, &appReceptor));

        /* Unimos los dispositivos al canal */
        p2pTx->Attach(canal);
        p2pRx->Attach(canal);

        /* Modificamos los parametros */
        canal->SetAttribute("Delay", TimeValue(e_retardo));
        p2pTx->SetDataRate(e_velocidad);

        /* Establecemos el tiempo de inicio y fin de la aplicacion transmisor */
        appTransmisor.SetStartTime(Seconds(1));
        appTransmisor.SetStopTime(Seconds(10));

        /* Corremos la simulacion */
        Simulator::Run();

        Simulator::Destroy();

        return appTransmisor.paquetesUtiles();

}
```

```
/*******************************************************************************
**
**      Fichero: auxiliar.cc
**
**      Autor: Belen Rodriguez Estevez
**
**      Descripcion: Archivo auxiliar donde se definen las aplicaciones
**                   transmisora y receptora.
**
**
*******************************************************************************/

/* Incluimos el fichero de cabecera */
#include "auxiliar.h"

/* Definimos el nombre para los registros de este fichero */
NS_LOG_COMPONENT_DEFINE("ArchivoAuxiliar");


/**************    Aplicacion Transmisora    ******************/

/* Constructor de la clase */
ClaseTransmisor::ClaseTransmisor(Ptr<NetDevice> t_dispositivo, Time t_temporizador,
                                 uint32_t t_tamPaquetes, uint8_t t_asentimiento)
{
        NS_LOG_FUNCTION(t_dispositivo << t_temporizador <<
                                         t_tamPaquetes << t_asentimiento);
        /* Identificamos el dispositivo transmisor */
        p2pTx = t_dispositivo;
```

```
	/* Establecemos el valor de los parametros */
	temporizador = t_temporizador;
	tamPaquetes = t_tamPaquetes;
	asentimiento = t_asentimiento;

	/* Iniciamos el numero de secuencia en cero*/
	secuencia = SequenceNumber16();

	/* Ponemos los contadores a cero */
	totalpaquetes = 0;
	totalACK = 0;
}


/* Se llama cuando empieza la simulacion */
void ClaseTransmisor::StartApplication()
{
	EnviarPaquete();
}


/* Se llama cuando ha terminado el tiempo de la simulacion */
void ClaseTransmisor::StopApplication()
{
	Simulator::Stop();
}


/* Funcion que envia un paquete con el siguiente numero de secuencia
  e inicia el temporizador de reenvio */
void ClaseTransmisor::EnviarPaquete()
{
	/* Creamos el paquete que vamos a enviar */
	Ptr<Packet> paquete = Create<Packet> (tamPaquetes);

	/* Anadimos la cabecera con el numero de secuencia correspondiente */
	MyHeader cabecera = MyHeader();
	cabecera.SetData (secuencia.GetValue());
	paquete->AddHeader (cabecera);


	NS_LOG_DEBUG("Numero de secuencia del paquete enviado: " << secuencia);
	/* Incrementamos el numero de secuencia para el proximo envio*/
	secuencia.operator++();

	/* Mandamos el paquete */
	p2pTx->Send(paquete, p2pTx->GetAddress(), 0x800);


	/* Incrementamos el numero de paquetes enviados */
	totalpaquetes++;

	/* Programamos el temporizador, lo iniciamos a t_temporizador y, si vence el tiempo,
	  llama a la funcion VenceTemporizador*/
	if(temporizador!=0)
		cuentaAtras = Simulator::Schedule(temporizador,&ClaseTransmisor::VenceTemporizador,this);
}
```

```
/* Cuando recibe un paquete, procesa el contenido de este */
bool ClaseTransmisor::ACKRecibido(Ptr<NetDevice> receptor, Ptr <const Packet> paquete,
                                  uint16_t protocolo, const Address & desde)
{
        NS_LOG_FUNCTION(receptor << paquete << protocolo << desde);

        uint8_t contenido=0;

        /* Obtenemos el contenido del paquete */
        paquete->CopyData(&contenido, 1);

        /* Si coincide con el codigo de asentimiento, cancelamos el temporizador
          y procedemos al envio del siguiente paquete */
        if (contenido == asentimiento)
        {
                NS_LOG_DEBUG("Asentimiento recibido");
                Simulator::Cancel(cuentaAtras);
                totalACK++;
                EnviarPaquete();
        }

return true;
}


/* Es llamada cuando la cuenta atras del temporizador expira*/
void ClaseTransmisor::VenceTemporizador()
{
        NS_LOG_WARN("Vence el temporizador");

        /* Decrementamos el numero de secuencia para reenviar el paquete anterior */
        secuencia.operator--();
        EnviarPaquete();
}


/* Metodo que devuelve el numero de paquetes utiles de la tranmision */
double ClaseTransmisor::paquetesUtiles ()
{
        NS_LOG_INFO("Porcentaje paquetes utiles: " << double(totalACK)/totalpaquetes*100 << "%");

return double(totalACK)/totalpaquetes*100;
}



/***************    Aplicacion Receptora    ******************/

/* Constructor de la clase */
ClaseReceptor::ClaseReceptor(Ptr<NetDevice> r_dispositivo, uint8_t r_asentimiento)
{
        NS_LOG_FUNCTION(r_dispositivo << r_asentimiento);
        /* Identificamos el dispositivo receptor */
        p2pRx = r_dispositivo;

        /* Establecemos el valor de los parametros */
        asentimiento = r_asentimiento;
}
```

```
/* Cuando recibe un paquete, envia un acuse de recibo */
bool ClaseReceptor::PaqueteRecibido(Ptr<NetDevice> receptor, Ptr <const Packet> paquete,
                                    uint16_t protocolo, const Address & desde)
{
        /* Hemos recibido un paquete */
        NS_LOG_DEBUG("Paquete Recibido");

        /* Creamos y enviamos el ACK */
        Ptr<Packet> paqueteAck = Create<Packet> (&asentimiento, 1);
        p2pRx->Send(paqueteAck, p2pRx->GetAddress(), 0x800);

return true;
}



/********************  Cabecera  ************************/

 MyHeader::MyHeader ()
 {
  // we must provide a public default constructor,
  // implicit or explicit, but never private.
 }
 MyHeader::~MyHeader ()
 {
 }

 TypeId
 MyHeader::GetTypeId (void)
 {
  static TypeId tid = TypeId ("ns3::MyHeader")
    .SetParent<Header> ()
    .AddConstructor<MyHeader> ()
  ;
  return tid;
 }
 TypeId
 MyHeader::GetInstanceTypeId (void) const
 {
  return GetTypeId ();
 }

 void
 MyHeader::Print (std::ostream &os) const
 {
  // This method is invoked by the packet printing
  // routines to print the content of my header.
  //os << "data=" << m_data << std::endl;
  os << "data=" << m_data;
 }
 uint32_t
 MyHeader::GetSerializedSize (void) const
 {
  // we reserve 2 bytes for our header.
  return 2;
 }
```

```
 void
 MyHeader::Serialize (Buffer::Iterator start) const
 {
  // we can serialize two bytes at the start of the buffer.
  // we write them in network byte order.
  start.WriteHtonU16 (m_data);
 }
uint32_t
MyHeader::Deserialize (Buffer::Iterator start)
{
 // we can deserialize two bytes from the start of the buffer.
 // we read them in network byte order and store them
 // in host byte order.
 m_data = start.ReadNtohU16 ();

 // we return the number of bytes effectively read.
 return 2;
}

void
MyHeader::SetData (uint16_t data)
{
 m_data = data;
}
uint16_t
MyHeader::GetData (void) const
{
 return m_data;
}
```

```
/*****************************************************************************
**
**          Fichero: auxiliar.h
**
**          Autor: Belen Rodriguez Estevez
**
**          Descripcion: Cabecera de los demas archivos, con los ficheros
**                       de cabecera necesarios asi como la definicion de
**                       las distintas clases y funciones empleadas.
**
**
*****************************************************************************/

/* Incluimos los ficheros de cabecera del software */
#include <ns3/core-module.h>
#include <ns3/point-to-point-module.h>
#include <ns3/application.h>
#include <ns3/sequence-number.h>
#include <ns3/drop-tail-queue.h>
#include <ns3/error-model.h>
#include <ns3/gnuplot.h>


using namespace ns3;

/* Definicion de la funcion que crea el escenario e inicia la transmision*/
double escenario (double e_velocidad, Time e_retardo, Time e_temporizador, uint32_t e_tampaquetes,
                                        uint8_t e_asentimiento, double e_error);
```

```
/* Definicion de la aplicacion transmisora */
class ClaseTransmisor: public Application
{
public:
        /* Constructor de la clase */
        ClaseTransmisor(Ptr <NetDevice> t_dispositivo, Time t_temporizador,
                                                uint32_t t_tamPaquetes, uint8_t t_asentimiento);

        /* Funciones */
        void EnviarPaquete();
        bool ACKRecibido(Ptr<NetDevice> receptor, Ptr <const Packet> paquete,
                                                uint16_t protocolo, const Address & desde);
        void VenceTemporizador();
        double paquetesUtiles ();


private:

        void StartApplication();
        void StopApplication();

        /* Variables */
        Ptr<NetDevice> p2pTx;
        Time temporizador;
        uint32_t tamPaquetes;
        EventId cuentaAtras;
        int totalpaquetes;
        int totalACK;
        uint8_t asentimiento;
        SequenceNumber16 secuencia;

};


/* Definicion de la aplicacion receptora */
class ClaseReceptor: public Application
{
public:
        /* Constructor de la clase */
        ClaseReceptor(Ptr<NetDevice> r_dispositivo, uint8_t r_asentimiento);

        /* Funciones */
        bool PaqueteRecibido(Ptr<NetDevice> receptor, Ptr <const Packet> paquete,
                                                        uint16_t protocolo, const Address & desde);

private:
        /* Variables */
        Ptr<NetDevice> p2pRx;
        uint8_t asentimiento;
};


/* Definicion de la cabecera */
 class MyHeader : public Header
 {
```

```
public:

  MyHeader ();
  virtual ~MyHeader ();

  void SetData (uint16_t data);
  uint16_t GetData (void) const;

  static TypeId GetTypeId (void);
  virtual TypeId GetInstanceTypeId (void) const;
  virtual void Print (std::ostream &os) const;
  virtual void Serialize (Buffer::Iterator start) const;
  virtual uint32_t Deserialize (Buffer::Iterator start);
  virtual uint32_t GetSerializedSize (void) const;
private:
  uint16_t m_data;
};
```

Anexo B: Código de ejemplos

```
/*************************************************************
**
**          Fichero: ejemplo1.cc
**
**          Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include <ns3/core-module.h>

using namespace ns3;

void hola();
void adios();

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

        Simulator::Schedule(Seconds(10), &hola);
        Simulator::Schedule(Seconds(20), &adios);
        Simulator::Run();
        Simulator::Destroy();

return 0;
}

void hola ()
{
        printf("Hola mundo en %f s\n", Simulator::Now().GetSeconds());
}

void adios()
{
        std::cout << "Adios mundo en " << Simulator::Now().GetSeconds() << "s" << std::endl;
}
```

```
/*************************************************************
**
**          Fichero: ejemplo1-2.cc
**
**          Autor: Belen Rodriguez Estevez
**
**
*************************************************************/
```

```
#include <ns3/core-module.h>

using namespace ns3;

void adios();
void hola();

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

	Simulator::Schedule(Seconds(10), &hola);
	Simulator::Schedule(Seconds(20), &adios);
	Simulator::Stop(Seconds(15));
	Simulator::Run();
	Simulator::Destroy();

return 0;
}

void hola ()
{
	std::cout << "Hola mundo en " << Simulator::Now().GetSeconds() << "s" << std::endl;
}

void adios()
{
	std::cout << "Adios mundo en " << Simulator::Now().GetSeconds() << "s" << std::endl;
}
```

```
/*************************************************************
**
**	Fichero: ejemplo1-3.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include <ns3/core-module.h>

using namespace ns3;

void hola();
void adios();

int main(int argc, char*argv[])
{
	Time::SetResolution(Time::S);

	Simulator::Schedule(Seconds(10), &hola);
	Simulator::Schedule(Seconds(20), &adios);
	Simulator::Run();
	Simulator::Destroy();

return 0;
}
```

```
void hola ()
{
	std::cout << "Hola mundo en " << Simulator::Now() << std::endl;
	Simulator::Stop();
}

void adios()
{
	std::cout << "Adios mundo en " << Simulator::Now() << std::endl;
}
```

```
/*****************************************************************
**
**	Fichero: ejemplo1-4.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/

#include <ns3/core-module.h>

using namespace ns3;

void hola();
void adios();
void cancelar();

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

	Simulator::Schedule(Seconds(10), &hola);
	Simulator::Schedule(Seconds(20), &adios);
	EventId id = Simulator::Schedule(Seconds(15), &cancelar);
	Simulator::Cancel(id);
	Simulator::Run();
	Simulator::Destroy();

return 0;
}

void hola ()
{
	std::cout << "Hola mundo en " << Simulator::Now().GetSeconds() << "s" << std::endl;
}

void adios()
{
	std::cout << "Adios mundo en " << Simulator::Now().GetSeconds() << "s" << std::endl;
}


void cancelar()
{
	std::cout << "No se debe ejecutar" << std::endl;

}
```

```
/*****************************************************************
**
**          Fichero: ejemplo2.cc
**
**          Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/

#include <ns3/core-module.h>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("ejemplo2");

void hola();
void adios();
void cancelar();

int main(int argc, char*argv[])
{
        Time::SetResolution(Time::S);

        Simulator::Schedule(Seconds(10), &hola);
        Simulator::Schedule(Seconds(20), &adios);
        EventId id = Simulator::Schedule(Seconds(15), &cancelar);
        Simulator::Cancel(id);
        Simulator::Run();
        Simulator::Destroy();

return 0;
}

void hola ()
{
        NS_LOG_DEBUG("Ha entrado en la funcion 'hola'");
}

void adios()
{
        NS_LOG_INFO("Adios mundo en " << Simulator::Now());
}


void cancelar()
{
        NS_LOG_ERROR("No puede entrar en esta funcion");
}
```

```
/*****************************************************************
**
**          Fichero: ejemplo2-2.cc
**
**          Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/
```

```
#include <ns3/core-module.h>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("ejemplo2");

void hola();
void adios();
void cancelar();

int main(int argc, char*argv[])
{
	Time::SetResolution(Time::S);

	LogComponentEnable("ejemplo2", LOG_LEVEL_INFO);
	Simulator::Schedule(Seconds(10), &hola);
	Simulator::Schedule(Seconds(20), &adios);
	EventId id = Simulator::Schedule(Seconds(15), &cancelar);
	Simulator::Cancel(id);
	Simulator::Run();
	Simulator::Destroy();

return 0;
}

void hola ()
{
	NS_LOG_DEBUG("Ha entrado en la funcion 'hola'");
}


void adios()
{
	LogComponentDisableAll(LOG_ALL);
	NS_LOG_INFO("Adios mundo en " << Simulator::Now());
}

void cancelar()
{
	LogComponentEnable("ejemplo2", LOG_ERROR);
	NS_LOG_ERROR("No puede entrar en esta funcion");
}
```

```
/*************************************************************
**
**	Fichero: ejemplo3.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include <ns3/core-module.h>

using namespace ns3;
```

```
NS_LOG_COMPONENT_DEFINE("ejemplo3");


int main(int argc, char*argv[])
{
	int dia=1;
	char mes[12] = "marzo";


	CommandLine cmd;

	cmd.Usage("ejemplo3.cc : Como introducir parametros por la linea de comandos.");
	cmd.AddValue("dia", "Dia en el que estamos", dia);
	cmd.AddValue("mes", "Mes en el que estamos", mes);
	cmd.Parse(argc, argv);

	std::cout << "Hoy es dia "<< dia << " de " << mes << std::endl;

	Simulator::Run();
	Simulator::Destroy();

return 0;
}
```

```
/*****************************************************************
**
**	Fichero: ejemplo4.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/

#include <ns3/core-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
	double media = 1;
	double varianza = 0.1;
	double intervalo = 0.25;
	int i;


	Ptr<NormalRandomVariable> var = CreateObject<NormalRandomVariable>();

	var->SetAttribute("Mean", DoubleValue(media));
	var->SetAttribute("Variance", DoubleValue(varianza));
	var->SetAttribute("Bound", DoubleValue(intervalo));

	for (i=0; i<=10; i++)
		std::cout << var->GetValue() << std::endl;

return 0;
}
```

```
/*****************************************************************
**
**          Fichero: ejemplo4-2.cc
**
**          Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/

#include <ns3/core-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
          double media = 1;
          double varianza = 0.1;
          double intervalo = 0.25;
          int i;

          RngSeedManager::SetSeed(2);
          Ptr<NormalRandomVariable> var = CreateObject<NormalRandomVariable>();

          var->SetAttribute("Mean", DoubleValue(media));
          var->SetAttribute("Variance", DoubleValue(varianza));
          var->SetAttribute("Bound", DoubleValue(intervalo));

          for (i=0; i<=10; i++)
                    std::cout << var->GetValue() << std::endl;

return 0;
}
```

```
/*****************************************************************
**
**          Fichero: ejemplo5.cc
**
**          Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/



#include <ns3/core-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
          double minimo = 1;
          double maximo = 5;
          int i;


          Ptr<UniformRandomVariable> var = CreateObject<UniformRandomVariable>();
```

```
	var->SetAttribute("Min", DoubleValue(minimo));
	var->SetAttribute("Max", DoubleValue(maximo));

	for (i=0; i<=10; i++)
		std::cout << var->GetValue() << std::endl;

return 0;
}
```

```
/*************************************************************
**
**	Fichero: ejemplo6.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include <ns3/core-module.h>
#include <ns3/gnuplot.h>

using namespace ns3;

int main(int argc, char*argv[])
{
	double media = 1;
	double varianza = 0.1;
	double intervalo = 0.25;
	double valor;
	int i;

	Gnuplot grafica("ejemplo6.png");
	grafica.SetTitle("ejemplo6.cc");
	grafica.SetLegend("Muestras", "Variable aleatoria normal");
	grafica.AppendExtra ("set xrange [-1:11]");
	grafica.AppendExtra("set yrange [0:1.5]");

	Gnuplot2dDataset curva;
	curva.SetTitle("Variable Normal");
	curva.SetStyle(Gnuplot2dDataset::LINES_POINTS);


	Ptr<NormalRandomVariable> var = CreateObject<NormalRandomVariable>();


	var->SetAttribute("Mean", DoubleValue(media));
	var->SetAttribute("Variance", DoubleValue(varianza));
	var->SetAttribute("Bound", DoubleValue(intervalo));


	for (i=0; i<=10; i++)
	{
		valor=var->GetValue();
		std::cout << valor << std::endl;
		curva.Add(i, valor);
	}
```

```
        grafica.AddDataset(curva);
        std::ofstream fichero ("ejemplo6.plt");
        grafica.GenerateOutput (fichero);
        fichero.close ();


return 0;
}
```

```
/*************************************************************
**
**      Fichero: ejemplo7.cc
**
**      Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include <ns3/core-module.h>
#include <ns3/simulator.h>
#include <ns3/nstime.h>

using namespace ns3;

void funcion(Time tiempo);

int main(int argc, char*argv[])
{
        Time temp = Time("10s");

        Ptr<EventImpl> evento = MakeEvent(&funcion, temp);

        Simulator::Schedule(temp, evento);
        Simulator::Run();
        Simulator::Destroy();

return 0;
}

void funcion(Time tiempo)
{
        std::cout << "Funcion llamada en: " << tiempo.GetSeconds() << "s" << std::endl;
}
```

```
/*************************************************************
**
**      Fichero: ejemplo8.cc
**
**      Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include <ns3/core-module.h>
#include <ns3/packet.h>
```

```
using namespace ns3;

int main(int argc, char*argv[])
{
	/* Transmision */

	uint8_t buffer = 255;

	Ptr<Packet> paquete = Create<Packet> (&buffer, 1);


	/* Recepcion */

	uint8_t contenido=0;

	paquete->CopyData(&contenido, paquete->GetSize());

	std::cout << "Recibido: " << (int)contenido << std::endl;

return 0;
}
```

```
/*************************************************************
**
**	Fichero: ejemplo9.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include "ns3/ptr.h"
#include "ns3/packet.h"
#include "ns3/header.h"
#include <iostream>

using namespace ns3;

/* A sample Header implementation
*/
class MyHeader : public Header
{
public:

 MyHeader ();
 virtual ~MyHeader ();

 void SetData (uint16_t data);
 uint16_t GetData (void) const;

 static TypeId GetTypeId (void);
 virtual TypeId GetInstanceTypeId (void) const;
 virtual void Print (std::ostream &os) const;
 virtual void Serialize (Buffer::Iterator start) const;
 virtual uint32_t Deserialize (Buffer::Iterator start);
 virtual uint32_t GetSerializedSize (void) const;
private:
 uint16_t m_data;
```

```
};

MyHeader::MyHeader ()
{
  // we must provide a public default constructor,
  // implicit or explicit, but never private.
}
MyHeader::~MyHeader ()
{
}

TypeId
MyHeader::GetTypeId (void)
{
 static TypeId tid = TypeId ("ns3::MyHeader")
   .SetParent<Header> ()
    .AddConstructor<MyHeader> ()
  ;
  return tid;
}
TypeId
MyHeader::GetInstanceTypeId (void) const
{
  return GetTypeId ();
}

void
MyHeader::Print (std::ostream &os) const
{
  // This method is invoked by the packet printing
  // routines to print the content of my header.
  //os << "data=" << m_data << std::endl;
  os << "data=" << m_data;
}
uint32_t
MyHeader::GetSerializedSize (void) const
{
  // we reserve 2 bytes for our header.
  return 2;
}
void
MyHeader::Serialize (Buffer::Iterator start) const
 {
   // we can serialize two bytes at the start of the buffer.
   // we write them in network byte order.
   start.WriteHtonU16 (m_data);
 }
uint32_t
MyHeader::Deserialize (Buffer::Iterator start)
{
  // we can deserialize two bytes from the start of the buffer.
  // we read them in network byte order and store them
  // in host byte order.
  m_data = start.ReadNtohU16 ();

  // we return the number of bytes effectively read.
  return 2;
}
```

```
void
MyHeader::SetData (uint16_t data)
{
  m_data = data;
}
uint16_t
MyHeader::GetData (void) const
{
  return m_data;
}


int main(int argc, char*argv[])
{
	/* Transmision */

	Ptr<Packet> paquete = Create<Packet> (1);
	MyHeader cabecera = MyHeader();
	cabecera.SetData (27913);
	paquete->AddHeader (cabecera);

	/* Recepcion */
	MyHeader recibido = MyHeader();

	paquete->RemoveHeader(recibido);

	recibido.Print(std::cout);
	std::cout << "\nRecibido: " << recibido.GetData () << std::endl;

return 0;
}
```

```
/*****************************************************************
**
**	Fichero: ejemplo10.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/

#include <ns3/core-module.h>
#include <ns3/sequence-number.h>

using namespace ns3;

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

	uint16_t dato=3;

	SequenceNumber<uint16_t,int16_t> secuencia = SequenceNumber<uint16_t,int16_t>(dato);
	SequenceNumber16 secuencia2 = SequenceNumber16(secuencia);

	std::cout << "Secuencia 1: " << (int)secuencia.GetValue() << std::endl;
	std::cout << "Secuencia 2: " << (int)secuencia2.GetValue() << std::endl;
```

```
return 0;
}
```

```
/*************************************************************
**
**      Fichero: ejemplo10-2.cc
**
**      Autor: Belen Rodriguez Estevez
**
**
*************************************************************/

#include <ns3/core-module.h>
#include <ns3/sequence-number.h>

using namespace ns3;


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

        uint8_t dato = 3;

        SequenceNumber<uint8_t,int8_t> secuencia =  SequenceNumber<uint8_t,int8_t>();
        SequenceNumber8 auxiliar;

        std::cout << "Valor inicial de la secuencia: "  << (int)secuencia.GetValue() << std::endl;

        auxiliar=secuencia.operator+(dato);
        std::cout << "Valor de la secuencia: " << (int)secuencia.GetValue();
        std::cout << ", resultado de sumar con " << (int)dato << " es: " << (int)auxiliar.GetValue()<< std::endl;

        std::cout << "Valor de la secuencia: "  << (int)secuencia.GetValue();
        std::cout << ", incrementar con prefijo: " << (int)secuencia.operator++().GetValue() << std::endl;

        std::cout << "Valor de la secuencia: " << (int)secuencia.GetValue();
        std::cout << ", incrementar con sufijo: " << (int)secuencia.operator++(dato).GetValue();
        std::cout << ", valor a posteriori: " << (int)secuencia.GetValue() << std::endl;


        if (secuencia.operator>(auxiliar))
                std::cout  << "La secuencia original es mayor que la auxiliar" << std::endl;
        else
                std::cout << "La secuencia original es menor o igual que la auxiliar"  << std::endl;

return 0;
}
```

```
/*************************************************************
**
**      Fichero: ejemplo11.cc
**
**      Autor: Belen Rodriguez Estevez
**
**
*************************************************************/
```

```
#include <ns3/core-module.h>
#include <ns3/error-model.h>
#include <ns3/packet.h>

using namespace ns3;

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

	int i;
	double contador =0;
	double contadorerr =0;

	Ptr<RateErrorModel> error = CreateObject<RateErrorModel> ();
	error->SetUnit(RateErrorModel::ERROR_UNIT_PACKET);
	error->SetRate (0.05);

	if (!error->IsEnabled())
		std::cout << "El modelo de error esta desactivado."  << std::endl;

	for(i=1; i<=1e6; i++)
	{
		Ptr<Packet> paquete = Create<Packet>();
		if (error->IsCorrupt(paquete))
			contadorerr++;
		contador++;
	}

	std::cout << "Tasa de error: " << contadorerr/contador*100   << "%"<< std::endl;

return 0;
}
```

```
/*****************************************************************
**
**	Fichero: ejemplo12.cc
**
**	Autor: Belen Rodriguez Estevez
**
**
*****************************************************************/

#include <ns3/core-module.h>
#include <ns3/point-to-point-module.h>

using namespace ns3;

int main(int argc, char*argv[])
{
	Time tiempo("1ms");
	DataRate velocidad ("3Mbps");

	NodeContainer nodos;
	nodos.Create(2);

	PointToPointHelper comunicacion;
	comunicacion.SetChannelAttribute("Delay", TimeValue(tiempo));
	comunicacion.SetDeviceAttribute("DataRate", DataRateValue(velocidad));
```

```
        comunicacion.Install(nodos);

return 0;
}
```