Capítulo 4

Lenguajes de OMNET++

Los programas de simulación son desarrollados en diferentes lenguajes, los cuales le permiten al programador definir los comportamientos de las simulaciones. En OMNeT++ se utilizan dos tipos de lenguaje: el primero de ellos, es desarrollado para implementar la parte gráfica de OMNeT++, su nombre es NED; el segundo, es utilizado para desarrollar la parte lógica del proyecto, C++. En este capítulo, se explican ambos lenguajes y algunas características necesarias para lograr una implementación en OMNeT++.

El lenguaje NED es una de las principales características de OMNeT++, ya que es quien le permite al usuario describir la estructura del modelo de simulación; en otras palabras, el lenguaje NED se utiliza para la descripción de las redes. Con este grupo de reglas sintácticas y semánticas es posible declarar módulos simples, los cuales representan elementos de la red, y módulos compuestos, que son grupos de módulos simples que trabajan de manera conjunta. También es posible referirse a la red como un módulo compuesto.

A su vez, los canales no son módulos, sino otro tipo de componente, y pueden ser utilizados para conectar módulos simples dentro de módulos compuestos.

4.1. Características del lenguaje NED

El lenguaje NED permite la escalabilidad de las redes simuladas, gracias a algunas de sus características (OMNeT++ Community, 2011a), como son:

Jerarquías. OMNeT++ es una herramienta que se centra en la simplicidad para crear los componentes; por esto, cada componente que es demasiado complejo puede ser dividido en varios módulos simples y ser representado como un módulo compuesto, lo que le permite al programador, editar con más facilidad cada uno de los elementos a utilizar y hacer modificaciones de una forma más sencilla.

Basado en componentes. Tanto los módulos simples como los compuestos son, de manera innata, reusables, lo que permite utilizar módulos desarrollados en cualquier otra parte de los proyectos, sin la necesidad de tener que copiar códigos. Gracias a esto, todos los programas desarrollados se convierten en librerías que pueden ser utilizadas por otros.

Interfaces. En cada módulo simple es posible definir las conexiones que este puede tener; a su vez, cada módulo compuesto tiene las conexiones que define el módulo simple que hace la función de interfaz. Como ejemplo de esto, se hace la suposición de un *router* (módulo compuesto) definido por dos módulos simples, una cola y un módulo de enrutamiento. El módulo de enrutamiento sería quien tiene la función de interfaz de salida, para comunicarse con los nodos hacia donde se puede encaminar la información. De modo análogo, la cola funcionaría como interfaz, pero esta vez de los paquetes entrantes; por ende, el módulo *router* (compuesto) tiene: las interfaces de entrada, que provee la cola (un módulo simple), y las interfaces de salida, que posee el módulo enrutador (otro módulo simple).

Herencia. Los módulos y los canales pueden tener subclases. Por ser una herramienta de simulación basada en componentes, como se dijo, tiene la posibilidad de implementar proyectos ya realizados, y de que a su vez, estos puedan ser reimplementados, dando la posibilidad de agregar parámetros, puertos y, en caso de ser módulos compuestos, de cambiar los módulos simples por otros.

Paquetes. El lenguaje NED presenta una estructura de paquetes, como la de *java*, para minimizar los riesgos de conflictos entre los diferentes paquetes. A su vez, utiliza, análogo a *Classpath* de *java, Nedpath*, que se generó para especificar la dependencia de los módulos de simulación, de una manera más sencilla.

4.2. Módulos simples

Cuando se hace referencia a un módulo simple, de manera implícita se puntualiza sobre los componentes activos del modelo. Un módulo simple se define por una palabra clave y es representado como un objeto para ser llamado por otras clases.

4.2.1. Simulaciones de eventos discretos

Un sistema de eventos discretos es un sistema donde los estados cambian; para el caso de OMNeT++, los eventos cambian y esto sucede en un instante de tiempo. Además, cada evento sucede en un tiempo cero (0), asumiendo que nada interesante sucede entre dos eventos consecutivos, lo que conlleva a representar la simulación de sistemas continuos.

Esto es ideal para un simulador de red, ya que generalmente estas tienen comportamientos de eventos discretos, como son: el inicio en la transmisión de un paquete, la llegada del paquete, el *timeout* de una retransmisión, etc.

El simulador utiliza una estructura de datos para hacer toda la simulación; al iniciar, todos los datos a simular se cargan en una estructura de datos llamada FES (*Future Event Set*), de la cual se extraen todos los eventos hasta que esta se encuentra vacía. Su pseudocódigo se presenta en la Figura 33.

```
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    ...
}
```

Figura 33. Comportamiento FES

4.2.2. Implementación de la FES

Esta implementación es de gran importancia para el rendimiento de las simulaciones de eventos discretos. Su funcionamiento en OMNeT++ es implementado como una pila binaria (*binary heap*). Su definición se encuentra en la clase *cMessageHeap*, pero generalmente es irrelevante para el programador de las simulaciones.

4.3. Estructura de los componentes y las conexiones

Los módulos desarrollados en OMNeT++, como se ha mencionado, se centran en módulos simples o compuestos y en conexiones que permiten el paso de mensajes entre ellos, de manera general (OMNeT++ Community, 2011a); las conexiones vienen con ancho de banda infinito, o canales ideales, lo que debe ser editado por el programador para generar el canal deseado.

Ya que los módulos simples son los componentes activos de la red, a su vez los canales también son entendidos como componentes, y ambos obedecen a códigos programados en lenguaje C++.

Teniendo en cuenta lo anterior, OMNeT++ diseñó todos los componentes heredados de la clase *cComponent*, un objeto que contiene todos los comportamientos de la clase *cObject*, ya que hereda de ella, como se presenta en la Figura 34.

También, se puede apreciar, que del cComponent se heredan dos clases: *cChannel* y *cModule*, que representan los canales y los módulos simples, respectivamente.

La clase *cChannel*, como se aprecia en la Figura 35, permite crear tres tipos de canales: *cIdealChannel*, *cDelayChannel* y *cDatarateChannel*. Aun así, el usuario tiene la posibilidad de crear su propia clase de canal, la cual, debe heredar de *cChannel* o de sus hijos directos.

Con la clase *cModule* ocurre algo similar; hereda dos clases principales, de las que ya se ha hablado: *cSimpleModule*, de la cual los usuarios generan aplicaciones, enrutadores

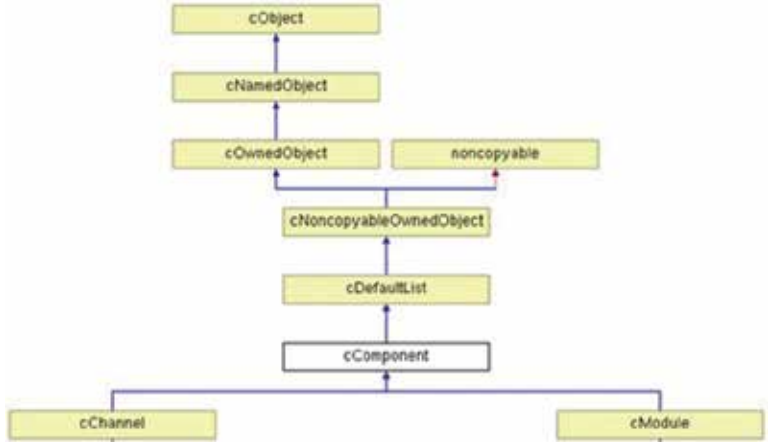

Figura 34. Representación *cComponent* (OMNeT Community, 2011b)

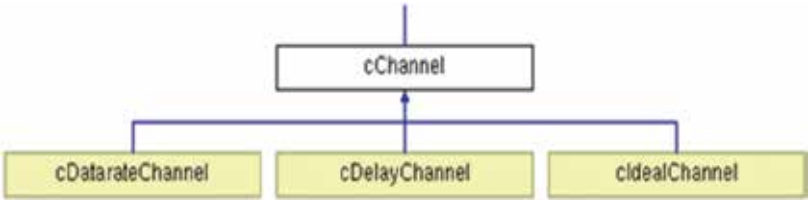

Figura 35. Herencia *cChannel* (OMNeT Community, 2011b)

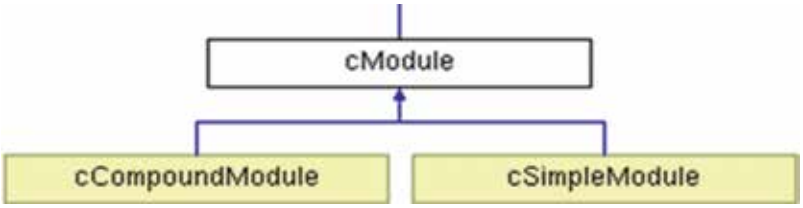

Figura 35. Herencia *cModule* (OMNeT Community, 2011b)

y nodos activos de la red; y *cCompoundModule*, que representan la unión de varios *cSimpleModule* para que funcionen en conjunto. Esto se puede ver en la representación de herencias que presenta la Figura 36.

Una vez entendidas las clases, se puede empezar a indagar sobre los métodos básicos que se utilizan en las simulaciones. Los cuatro métodos principales son:

» *Initialize()*.Se encarga de dar los valores iniciales a la simulación y de iniciar el resto del programa.

» *Finish()*.Se encarga de terminar el código correctamente. El método es llamado automáticamente cuando la FES termina pero, si se detiene la simulación, es recomendable llamar este método para minimizar los posibles errores de compilación.

» *Handle Message (cMessage *msg)*. Es invocado con el mensaje como parámetro, siempre que el módulo recibe un mensaje. Este método es el encargado de procesar el mensaje y de retornarlo a la simulación; el tiempo transcurrido en el método no se asume como tiempo de simulación.

» *Activity()*. El método empieza como un hilo al iniciar la simulación y corre hasta que termina o hasta que se utilice *return*. Los mensajes son obtenidos con el método *receive()* y el tiempo de simulación se toma cuando actúa el método *receive()*.

4.4. Definir módulos simples en OMNet++

Como se ha mencionado, un módulo simple es un programa en C++ que, básicamente, hereda de la clase *cSimpleModule*. Para conectar la programación en lenguajes C++ y NED, es necesario hacerlo por medio del método *Define_Module()*, el cual se encarga de registrar la clase C++ con OMNeT++.

» Un módulo simple debe tener definidas como prioritarios los siguientes elementos:

» un *include files*, que incluya la cabecera de *omnetpp.h*;

» un modelo que virtualiza las clases y define el tipo de módulo a utilizar, como se presenta en la Figura 37;

» el registro de la clase .cc a OMNeT++, por medio de *define_module()*; y

» la definición de los módulos incluidos y virtualizados en la cabecera.

```
classNombre_Modulo : publiccSimpleModule
{
protected:

      virtualvoidinitialize();
      virtualvoidhandleMessage(cMessage *msg);
      virtualvoidactivity();
   };
```

Figura 37. Estructura inicial módulo simple

4.5. Constructor

Los módulos simples nunca son instanciados por los usuarios que crean las clases, ya que todos los módulos son heredados directamente de la clase *cSimpleModule*. Lo anterior implica que los constructores no se pueden definir arbitrariamente. Para solucionar esto, OMNeT++ define para su implementación los constructores públicos. Los módulos simples no deben tener argumentos.

Por otro lado, la clase *cSimpleModule* presenta dos constructores: el primero de ellos, sin ningún tipo de argumentos, como se presenta en la Figura 38**;** el segundo, con la posibilidad de cambiar el tamaño de la pila, como se presenta en la Figura 39**.** Este último constructor es importante cuando se va a utilizar el método *activity()*, ya que el tamaño por defecto de la pila se agota al usar este método.

```
classCol : publiccSimpleModule
{
public:
      Col() : cSimpleModule(){}
      virtualvoidinitialize();
      virtualvoidactivity();
      virtualvoidhandleMessage(cMessage *msg);
};
```

Figura 38. Constructor sin argumentos

```
classCol : publiccSimpleModule
{
public:
      Col() : cSimpleModule(16384){}
      virtualvoidinitialize();
      virtualvoidactivity();
      virtualvoidhandleMessage(cMessage *msg);
};
```

Figura 39. Constructor con argumentos

Anexo a lo anterior, cabe decir que el lenguaje NED presta numerosos recursos adicionales, lo que ayuda a desarrollar más herramientas para la comprensión de la telemática y el funcionamiento de los sistemas en red. Lo que se ha presentado en este capítulo, responde a las necesidades básicas para lograr la implementación de una red, encaminado a entender la influencia de diferentes colas en la calidad de servicio (QoS).