ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
DE TELECOMUNICACIÓN UNIVERSIDAD
POLITÉCNICA DE CARTAGENA

Trabajo Fin de Grado

Simulador de redes vehiculares mediante integración de motores de videojuegos y simulador de redes

AUTOR: Nazareth Gómez Bueno
DIRECTOR: Esteban Egea López
Septiembre / 2015

Autor	Nazareth Gómez Bueno
E-mail del autor	nazarethgomezbueno@gmail.com
Director	Esteban Egea López
E-mail del director	esteban.egea@upct.es
Título del PFG	Simulador de redes vehiculares mediante integración de motores de videojuegos y simulador de redes
Descriptores	Redes vehiculares, simulación, sincronización

Resumen

Desarrollo de un simulador integrado de redes y tráfico para la simulación de situaciones de riesgo de accidente mediante integración de motor de videojuegos y simulador de redes. Nos centraremos en el desarrollo de librerías con los componentes necesarios para permitir el paso de mensajes y la sincronización entre ambos simuladores mediante sockets. El simulador podrá ser usado en un contexto de aprendizaje como demostrador de situaciones de riesgo.

Titulación	Grado en Ingeniería Telemática
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de presentación	Septiembre 2015

Resumen

Desarrollo de un simulador integrado de redes y tráfico para la simulación de situaciones de riesgo de accidente mediante integración de motor de videojuegos y simulador de redes. Nos centraremos en el desarrollo de librerías con los componentes necesarios para permitir el paso de mensajes y la sincronización entre ambos simuladores mediante sockets. El simulador podrá ser usado en un contexto de aprendizaje como demostrador de situaciones de riesgo.

Índice

Índice de figuras

1. Introducción

La utilidad real de algunos sistemas eSafety, en particular, los basados en tecnologías de comunicaciones y cooperativas, no está establecida de manera concluyente. Uno de los problemas a los que se enfrenta el desarrollo de estas tecnologías es la falta de herramientas de simulación adecuadas. En particular, la falta de simuladores de situaciones de riesgo realistas. Y sobre todo a un coste razonable y con medios disponibles para un alto número de potenciales investigadores.

En este proyecto se propone cubrir este hueco aprovechando las capacidades de simulación de entornos virtuales y física realistas de los motores de videojuegos de última generación que ahora son accesibles a coste cero. Además, una herramienta de estas características se puede utilizar en un contexto puramente educativo y de concienciación de los riesgos de la conducción.

El objetivo final, por tanto, es el desarrollo de un simulador integrado de redes y tráfico para la simulación de situaciones de riesgo de accidente mediante integración de motor de videojuegos y simulador de redes. El simulador podrá ser usado en un contexto de aprendizaje como demostrador de situaciones de riesgo.

Para poder llevar a cabo dicho propósito, este proyecto se va a centrar en el desarrollo de librerías con los componentes necesarios para permitir el paso de mensajes y la sincronización entre ambos simuladores mediante sockets.

La necesidad de realizar esa sincronización es con el fin de que ambos simuladores avancen en la simulación al mismo tiempo. Esto supone enfrentarse al problema del tiempo en cada simulador, ya que los simuladores de redes funcionan con eventos discretos y los motores de videojuegos en tiempo real. Por este motivo, hay que tener muy presente qué unidades de tiempo se van a utilizar para marcar la sincronización entre ambos.

Se integrará el simulador de redes OMNeT++ con un motor gráfico adecuado, en principio Unity.

2. Tecnologías utilizadas

Se han utilizado diferentes tecnologías para el desarrollo de este proyecto. A continuación se detallan cuales han sido y porque se han elegido.

2.1. Boost C++

Boost es un conjunto de librerías para la programación en C++ que da soporte a tareas y estructuras como álgebra lineal, generación de números pseudoaleatorios, multi-hilos, procesamiento de imagen, expresiones regulares y unidades de pruebas. Contiene unas ochenta librerías individuales.

Nosotros utilizaremos *Boost.Asio*, que es una librería multiplataforma en C++ orientada a redes y a la programación en bajo nivel de E/S (Entrada/Salida) haciendo uso de la nomenclatura del actual C++, siendo así más sencillo su uso.

2.2. FlatBuffers

FlatBuffers es una librería de serialización multiplataforma para C++, aunque soporta Java, C# y Go. Fue creado por Google y, gracias a su eficiencia, proporciona unas latencias[1] realmente bajas, perfecto para el desarrollo de juegos o aplicaciones de rendimiento crítico.

[1]Latencia es la suma de retardos temporales dentro de una red. Un retardo es producido por una demora en la propagación y transmisión de paquetes dentro de la red.

A través de un compilador, convertimos el fichero con código IDL[2] en cabeceras o clases (según lenguaje).

Se ha optado por utilizar *FaltBuffers* para abordar el problema de la comunicación entre ambos simuladores, ya que cada uno hace uso de un lenguaje de programación distinto. Para ello, se utiliza esta librería con el fin de poder llevar a cabo la serialización de la información entre plataformas.

2.3. OMNeT++

OMNeT++ es un framework y una librería de simulación extensible, modular y basada en componentes C++, principalmente desarrollado para construir simuladores de redes, ya sean cableadas, inalámbricas, o de cualquier otro tipo.

A su vez, *OMNeT++* ofrece un entorno de desarrollo basado en Eclipse, un entorno gráfico para el tiempo de ejecución, y otras herramientas. Tiene extensiones para la simulación en tiempo real, emulación de redes, integración de base de datos, integración de SystemC, y muchas otras funciones.

Proporciona una arquitectura en componentes para los modelos. Los componentes (*módulos*) están programados en C++, posteriormente se unen en componentes más grandes o modelos usando un lenguaje de alto nivel (*NED*, Network Description File). Gracias a la arquitectura modular de *OMNeT++*, el kernel de simulación (y modelos) pueden ser insertados en aplicaciones propias fácilmente.

Componentes

- Librería para la simulación del kernel.
- Lenguaje NED para la descripción de topologías.
- IDE de *OMNeT++* basado en la plataforma Eclipse.

[2]IDL (o *Interface Definition Language*) describe una interfaz en un lenguaje neutral, que permite la comunicación entre los componentes de software desarrollados en diferentes lenguajes de programación, como por ejemplo entre C++ y Java.

- Interfaz gráfica para la ejecución de simulaciones, con enlaces al ejecutable de simulación (Tkenv).
- Interfaz de línea de comandos para la ejecución de la simulación (Cmdenv).
- Utilidades (herramienta de creación de makefiles, etc.).
- Documentación, ejemplos de simulaciones, etc.

Plataformas

OMNeT++, así como su IDE, se puede ejecutar sobre Windows, Linux, Mac OS X, y cualquier otro sistema basado en UNIX.

2.4. Unity

Unity es un motor de videojuegos realista. Una plataforma de desarrollo flexible y poderoso para crear juegos y experiencias interactivos 3D y 2D multiplataforma.

A través de su editor *MonoDevelop*, permite desarrollar scripts en C#, UnityScript[3], Boo y JavaScript.

Plataformas

Unity está disponible como plataforma de desarrollo para Windows y Mac OS X, y permite crear juegos para Windows, Mac OS X, Linux, Xbox 360, PlayStation 3, PlayStation Vita, Wii, Wii U, iPad, iPhone, Android y Windows Phone.

Debido a su gran facilidad y flexibilidad en cuanto a la incorporación de scripts que interactuan con el medio virtual, además de contar con una versión gratuita muy completa, hacen de *Unity* un gran candidato a tener en cuenta en lo relacionado con simulaciones realistas, como es este caso.

[3]UnityScript es un lenguaje personalizado, propio de Unity, inspirado en la sintaxis ECMAScript.

2.4.1. UnityCar

Dentro de *Unity*, existe una gran variedad de paquetes que amplían su funcionalidad en gran medida, como es el caso de *UnityCar*.

UnityCar es paquete de simulación de vehículos realista y completo para *Unity*. Con *UnityCar* podemos integrar fácilmente cualquier tipo de vehículo completamente funcional en nuestro proyecto. Este paquete incluye una serie de escenas donde ver todo su potencial, además de una gran variedad de vehículos para probar, simulando su comportamiento real.

3. Desarrollo del proyecto

A continuación se encuentra el plan de desarrollo del proyecto, es decir, su arquitectura, qué partes lo componen y cómo se han solventado los problemas que han ido surgiendo.

3.1. Arquitectura general

Aunque estas tecnologías sean independientes, en este proyecto las hemos unificado como una sola para poder integrar el simulador propuesto, o al menos, una base sólida del mismo. A continuación se va a comentar como está estructurada la arquitectura sin matizar demasiado.

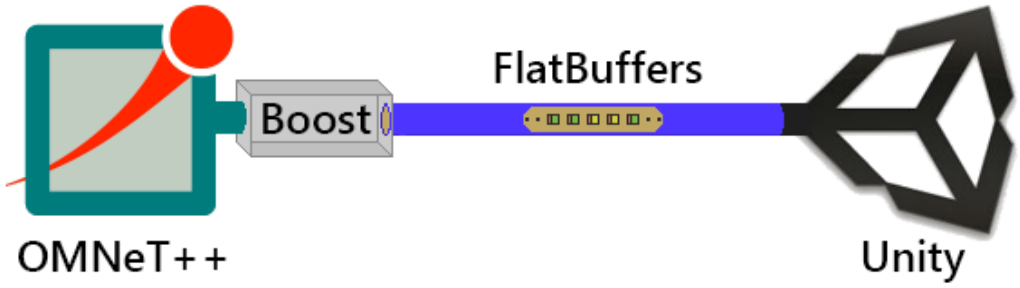


Figura 1: Arquitectura general

Profundizando un poco en cómo éstas tecnologías están relacionadas, el simulador de redes *OMNeT++* utiliza la librería de comunicaciones *Boost.Asio* para simplificar el uso del socket que se va a encargar de la comunicación con el medio.

⇒*OMNeT++* es un framework y una librería de simulación extensible, modular y basada en componentes C++, principalmente desarrollado para construir simuladores de redes, ya sean cableadas, inalámbricas, o de cualquier otro tipo.

⇒*Boost.Asio* es una librería perteneciente al conjunto de *Boost C++*, multiplaforma orientada a redes y a la programación en bajo nivel de E/S (Entrada/Salida), haciendo uso de la nomenclatura del actual C++.

En el otro extremo, el motor de videojuegos *Unity* utiliza las clases disponibles en C# para comunicarse con el medio.

⇒*Unity* es un motor de videojuegos realista. Una plataforma de desarrollo flexible y poderoso para crear juegos y experiencias interactivos 3D y 2D multiplataforma.

Esta comunicación es posible gracias a la librería *FlatBuffers* que se encarga de serializar y deserializar la información que se transmite, haciendo posible la comunicación entre ambos simuladores, indistintamente del lenguaje que use cada uno (C++ en *OMNeT++* y C# ó JavaScript en *Unity*).

⇒*FlatBuffers* es una librería de serialización multiplataforma para C++, aunque soporta Java, C# y Go. Fue creado por Google y, gracias a su eficiencia, proporciona unas latencias realmente bajas, perfecto para el desarrollo de juegos o aplicaciones de rendimiento crítico.

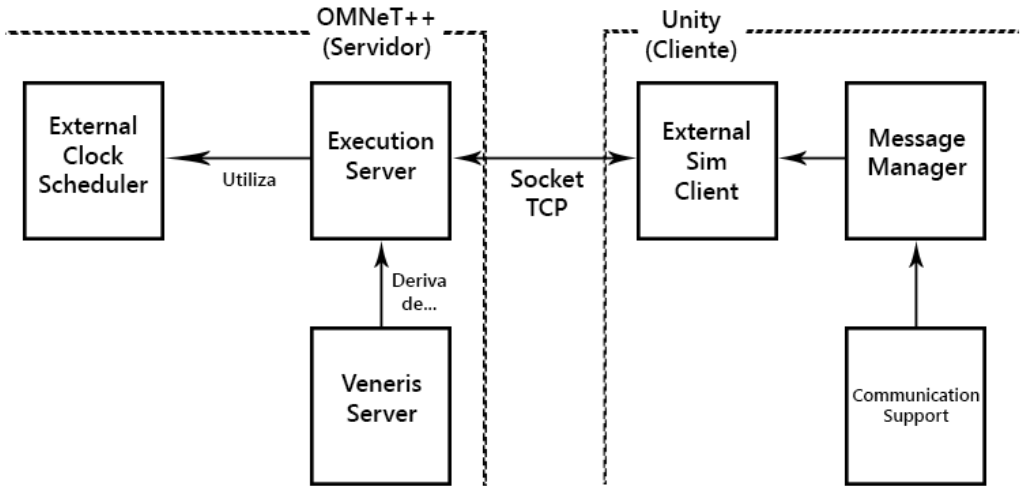


Figura 2: Diagrama general del simulador integrado

Si profundizamos aún más, podemos ver la estructura interna de los simuladores.

Por un lado tenemos *OMNeT++*, cuyo módulo principal y servidor es *VenerisServer*, específico para nuestro simulador, que deriva directamente del módulo *ExecutionServer*, que es el encargado de las comunicaciones. A su vez, se utiliza *ExternalClockScheduler* para la sincronización externa.

Por otro lado tenemos *Unity*, compuesto por tres scripts principales escritos en C#. El encargado de comunicarse con los componentes del proyecto es

CommunicationSupport. Éste le envía la información a *MessageManager*, que la guardará en una cola. Será, por tanto, *ExternalSimClient* el responsable de sacar la información de *MessageManager* y enviarla al medio.

La conexión se establece usando el protocolo TCP, con el fin de asegurar la recepción de la información.

3.2. Boost C++

Con el fin de simplificar el uso y gestión de la conexión en el servidor (*OMNeT++*), independientemente de la plataforma y haciendo uso de C++, se ha optado por utilizar la librería *Boost C++*, concretamente *Boost.Asio*.

En este caso, la plataforma usada ha sido Windows, por lo que se debe compilar *Boost C++* para Windows. Como entorno de compilación se ha utilizado *MinGW*.

Lo primero que se debe hacer es descargar e instalar *MinGW* desde su web oficial. Lo siguiente será descargar el código fuente de *Boost C++* y descomprimirlo en una carpeta, por ejemplo, en la ruta *C:\Boost_cpp_source*.

Una vez hecho esto, es hora de compilar. Para ello, vamos a la carpeta donde hemos descomprimido el código fuente de *Boost C++* y abrimos una ventana de comandos y escribimos lo siguiente:

```
bootstrap.bat mingw
```

Donde se debe indicar el entorno a usar (siendo Windows, sino se indica nada, tomará por defecto *msvc*).

Una vez finalice, escribimos lo siguiente:

```
b2 install -prefix=C:\Boost_cpp toolchain=gcc
```

Donde se indica en que directorio se va a guardar el resultado de la compilación. Además, se debe indicar el compilador a usar, ya que hemos usado el entorno *MinGW*, usaremos el compilador *gcc*. Guardamos el código fuente y el resultado de la compilación en carpetas separadas para evitar confundir

ficheros/directorios.

Una vez finalizados todos los pasos, *Boost C++* está listo para ser usado en nuestros proyectos.

3.3. FlatBuffers

Esta potente librería nos permite serializar la información para poder establecer una comunicación entre ambos simuladores y que se entiendan.

En resumen, permite la creación de una interfaz IDL común, indistintamente del lenguaje de programación que se utilice.

Antes de nada, deberemos bajar el código fuente y compilarlo, aunque en el propio repositorio existe el binarios compilado para usar en Windows, lo que nos ahorrará tiempo y esfuerzo. Por tanto, descargamos el fichero comprimido que contiene *FlatBuffers* compilado y lo descomprimimos en una carpeta, por ejemplo:

```
C:\FlatBuffers
```

Una vez lo tenemos descomprimido, podemos empezar a crear nuestros esquemas.

3.3.1. Tipos de mensajes

Para este proyecto, se han definido unos tipos de mensajes muy concretos según el funcionamiento requerido como base:

- Header
  Contiene el tipo de mensaje que viene a continuación y su longitud (en bytes).
- Create
  Indica al servidor que instancie un nuevo módulo representando un nuevo vehículo en la simulación. A su vez, indica el ID y la posición del vehículo (mediante una estructura Vec3[4]).

[4]Vec3 es una estructura que contiene las coordenadas (x,y,z) del vehículo.

- Destroy
  Indica al servidor que debe eliminar un módulo de la simulación, según el ID indicado.

- CAM
  Contiene la posición, la velocidad global, la velocidad local y el ID de un vehículo en cuestión.

- ExternalTime
  Contiene el tiempo interno del cliente para la sincronización con el servidor.

- End
  Indica al servidor que finalice la simulación.

Para definir estos mensajes se utiliza un lenguaje neutral muy similar a los lenguajes de la familia C (C, C++, C#,...), con el que se componen dichos esquemas. Un ejemplo de esquema, definiendo el mensaje *Header*:

```
1  namespace Simulator;
2
3  attribute "priority";
4
5  table Header{
6    type:uint;
7      // Tipos
8      //
9      // 0: Reserved
10     // 1: Create
11     // 2: Destroy
12     // 3: CAM (Mov, update POSITION)
13     // 4: ExternalTime
14     // 5: End
15     //
16   size:uint;
17 }
18
19 root_type Header;
```

Siempre se debe declarar un *namespace* en el que estarán todos los esquemas que generemos, así como el atributo *"priority"*, para evitar posibles problemas al pasarlo por el binario que genera el código que vamos a usar (el binario *flatc*).

El tipo *table* es un objeto que contiene los campos que definamos dentro, en este caso dos campos *unsigned integer*, uno para indicar el tipo de mensaje que viene a continuación y otro para indicar su longitud. Dicha tabla será lo que se reconstruya en el otro extremo una vez que se envíe serializado.

Una vez definido el esquema de nuestro mensaje, debemos pasarlo por el intérprete para que nos genere los correspondientes ficheros que usaremos en nuestro proyecto. Para ello, abrimos un terminal donde se encuentren nuestros esquemas y el binario *flatc* y escribimos:

Para C++

```
flatc -c esquema.idl
```

Esto generará un fichero denominado "esquema_generated.h", que contiene nuestro esquema interpretado para C++.

Para C#

```
flatc -n fichero.idl
```

Esto generará un fichero denominado "Esquema.cs", que contiene nuestro esquema interpretado para C#, o .Net, de ahí el argumento '-n'.

Tras generar todos los ficheros con nuestros tipos de mensajes según los esquemas que hayamos definido, podemos usarlos, junto a la librería *FlatBuffers*, en nuestro proyecto.

3.4. Unity

El motor gráfico de videojuegos *Unity* va a hacer el papel de cliente. Se encargará de enviar información, de forma continuada, hacia el servidor.

3.4.1. Inicialización

Para el desarrollo del cliente no vamos a realizar todo el trabajo desde cero, sino todo lo contrario.

Primero, vamos a la web oficial de *Unity* y descargamos la versión *Personal Edition*. Una vez descargado, lo instalamos y listo. En cuanto a *UnityCar*, es un paquete de pago, por lo que es necesario comprarlo en su web oficial.

Existen dos versiones: Pro y LE (Light Edition), nosotros usaremos la versión Pro.

Inicialmente partimos de un proyecto vacío en *Unity*, en el que cargamos el paquete *UnityCar*. Para ello vamos a:

```
Assets > Import Package > Custom Package...
```

y elegimos el paquete *UnityCar*, que en nuestro caso se llama *unitycar.unitypackage*.

Una vez que el proyecto base está listo, incluimos la carpeta *Simulation* generada mediante *FlatBuffers*, la cual incluye los ficheros para generar los mensajes necesarios para que exista la comunicación cliente-servidor.

Añadir que, el paquete *UnityCar* usa unos controles adicionales[1] a los que trae *Unity* por defecto, por tanto, hay que añadirlos. Para ello, en la interfaz de *Unity*, vamos al InputManager:

```
Edit > Project Settings > Input
```

Y lo dejamos como en la siguiente imagen.

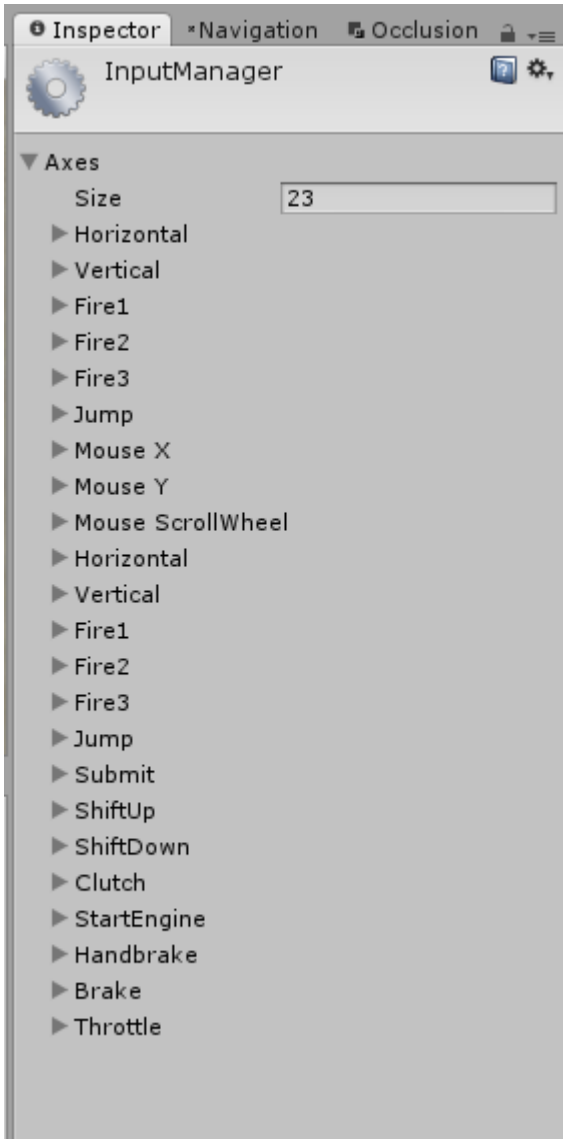


Figura 3: Configuración de los controles

Donde:

- **ShiftUp** es para subir marchas.
- **ShiftDown** es para bajar marchas.
- **Clutch** es el embrague.
- **StartEngine** es para arrancar el vehículo.

- **Handbrake** es el freno de mano.
- **Brake** es el freno.
- **Throttle** es el acelerador.

Una vez añadidos estos parámetros, los configuramos a nuestras necesidades. Una vez finalizado este proceso, ya es posible controlar los vehículos de *UnityCar*.

3.4.1.1. Configuración de FlatBuffers

Con el fin de poder utilizar los mensajes que creamos anteriormente, debemos incluir en nuestro proyecto la librería *FlatBuffers*, para poder darle un sentido a dichos mensajes.

Al descargar y descomprimir el fichero comprimido con el binario compilado, encontramos varias carpetas en su interior, pero la que nos interesa es la que se encuentra en la siguiente ruta:

```
C:\FlatBuffers\src\net
```

Ahí se encuentra la carpeta *FlatBuffers* que deberemos arrastrar e incluir en nuestro proyecto. Una vez hecho, basta con incluir lo siguiente en cada script que queramos usar los mensajes previamente generados:

```
1 using FlatBuffers;
2 using Simulator;
```

3.4.2. Script CommunicationSupport

Proporciona la conexión entre el paquete *UnityCar* y el resto de scripts encargados de la comunicación con el servidor, sin necesidad de conocer nada más que las funciones que implementa.

Contiene una clase estática, de tal forma que no hace falta instanciar ningún objeto referido a esta clase, sino que se pueden llamar a sus métodos directamente, haciendo su uso más sencillo y simple.

Contiene tres métodos:

CommunicationSupport

- void sendCarPosition(GameObject _car)
- void sendCreateCar(GameObject _car)
- void sendTime()

Estos métodos tienen la siguiente funcionalidad:

- void sendCarPosition(GameObject _car)

  Genera un mensaje de tipo *CAM* con los datos del GameObject indicado. Finalmente, encola el mensaje.

- void sendCreateCar(GameObject _car)

  Genera un mensaje del tipo *Create* con los datos del GameObject indicado. Finalmente, encola el mensaje.

- void sendTime()

  Genera un mensaje del tipo *ExternalTime* con el tiempo actual de la simulación ejecutándose en *Unity*.

Cuando hacemos mención a "encolar un mensaje" nos referimos a pasar el mismo a la cola localizada en el script *MessageManager*.

¿Por qué el cliente envía su tiempo interno al servidor?

Una bombilla no se enciende si no le damos al interruptor. Aplicando esta misma lógica, si no interactuamos con el vehículo, ¿qué mensajes vamos a procesar en el servidor?

Por este motivo, el encargado de la sincronización es el cliente, por tanto, es el cliente quien controla al servidor.

El tiempo interno de *Unity* es un tiempo decimal que se incrementa conforme avanza la simulación. Por otro lado, tenemos el simulador de redes *OMNeT++*, cuyo tiempo es por eventos discretos. Debido a esto, el cliente será quien indique al servidor hasta donde puede avanzar, indicándole su tiempo en todo momento como tope de avance en su simulación.

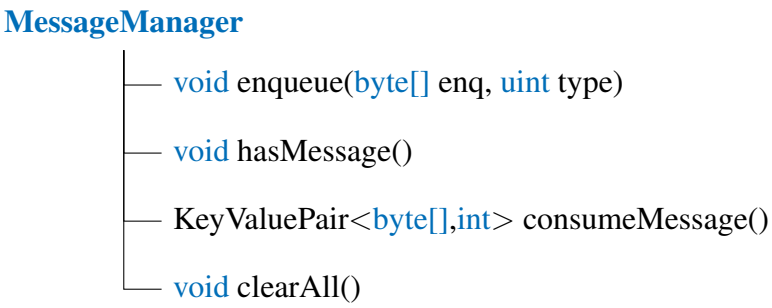
Esto quiere decir, que el servidor avanzará hasta un tiempo indicado, indistintatemente de que tenga más mensajes pendientes para procesar, hasta que no se actualice dicha marca temporal, no podrá continuar avanzando.

3.4.3. Script MessageManager

Se encarga de gestionar la cola FIFO[5] que guarda la información de los mensajes a enviar posteriormente. Concretamente, se guarda el mensaje en sí, y el tipo de mensaje. El formato en el que se guardar la información es mediante la estructura *KeyValuePair<TKey, TValue>*.

Contiene una clase estática, cuyos métodos son llamados en el script *CommnicationSupport*, por lo que son transparentes desde el punto de vista de *UnityCar*.

Contiene cuatro métodos:



Estos métodos tienen la siguiente funcionalidad:

- void enqueue(byte[] enq, uint type)

  Inserta al final de la cola el mensaje a enviar.

- void hasMessage()

  Devuelve *true* si la cola contiene algún mensaje para enviar. En cualquier otro caso, devolverá *false*.

- KeyValuePair<byte[],int> consumeMessage()

  Quita de la cola el primer mensaje y lo devuelve.

- void clearAll()

  Saca todos los mensajes de la cola, dejándola vacía.

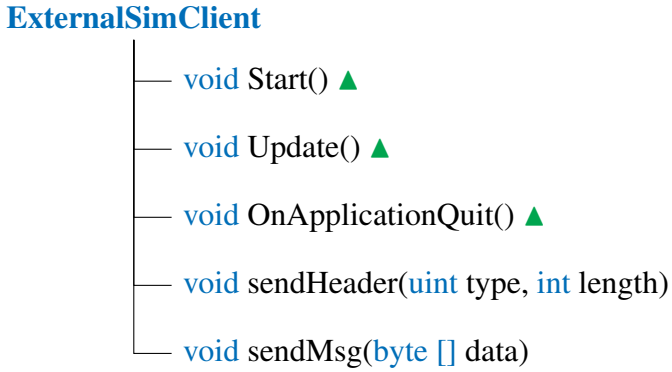
[5]FIFO (First In, First Out) va referido a una cola, en la que el primer elemento en entrar en ella, será el primer elemento en salir.

3.4.4. Script ExternalSimClient

Es el encargado de establecer un canal de comunicaciones con el servidor. Hereda directamente de la clase *MonoBehaviour* de *Unity*, con el fin de poder usar, sobre todo, el método *Update()*.

Al heredar de *MonoBehaviour*, el script necesita estar enlazado con un GameObject del entorno para que se ejecute[2]. En nuestros caso, lo hemos enlazado con el GameObject que representa la ciudad donde realizamos la simulación.

Contiene cinco métodos, de los cuales tres son heredados:



Estos métodos tienen la siguiente funcionalidad:

- void Start() ▲

  Este método se invoca automáticamente al iniciar la simulación, por tanto, será el encargado de inicializar el socket y establecer la comunicación, según una dirección (en este caso *localhost*) y un puerto indicado.

- void Update() ▲

  Se invoca una vez por frame, por lo que el número de veces que se invoque dependerá de los frames por segundo (FPS) a los que se esté ejecutando la simulación.

  Es el encargado de comprobar si existen mensajes para transmitir. De ser así, obtiene el mensaje a través de *MessageManager* y lo envía, con su correspondiente cabecera.

- void OnApplicationQuit() ▲

Se invoca cuando se cierra el entorno de la simulación. Utilizaremos este método para limpiar la cola de mensajes pendientes y de enviar un mensaje de tipo *End* para indicar al servidor que hemos terminado.

A su vez, una vez enviado dicho mensaje, cerramos la conexión, liberando el puerto usado.

- void sendHeader(uint type, int length)

  Genera y envía un mensaje de tipo *Header* (cabecera), incluyendo los valores indicados.

  Este mensaje se envía primero, y posteriormente, el mensaje que realmente nos interesa, con el fin de indicarle al servidor previamente el tipo de mensaje que va a llegar y su longitud.

- void sendMsg(byte [] data)

  Envía a través del socket el mensaje en cuestión, es decir, el formado por la cadena de bytes *data*.

En cuanto a la parte de comunicaciones, empleamos las clases *TcpClient*, para iniciar la conexión, y *NetworkStream*[6], para obtener el flujo donde escribir/leer del socket, una vez inicializado correctamente.

3.4.5. Script StartGame

Instancia y lanza la simulación según la escena elegida. En nuestro caso, elegimos la escena que de la ciudad.

Se encarga de instanciar, en caso de que no existan, los vehículos que se van a utilizar, así como posicionarlos en distintas localizaciones. Además, permite cambiar de vehículo usando las teclas **PageUp** y **PageDown**.

También controla las cámaras, es decir, el ángulo desde el que visualizamos el vehículo, en función del modo en el que nos encontremos.

La idea principal de los scripts creados para el cliente es intentar modificar lo menos posible de cualquier proyecto en el que se quiera incluir. Para hacer esto bien, tendríamos que crear el proyecto en función de nuestros scripts, para adaptar bien nuestro código al de los scripts y simplificar las cosas, pero esto no es viable sobre proyectos ya creados. Por tanto, se han creado para que las

modificaciones sean mínimas y sencillas.

Principalmente, nos interesan tres puntos:

1. Saber cuando se instancian los vehículos, o en su defecto, cuando se activan (en caso de estar ya instanciados).
2. Conocer, de forma periódica, el tiempo interno de *Unity*.
3. Conocer, de forma periódica, la posición de los vehículos.

Una vez conocido esto, sólo tenemos que ver el script y ubicar lo que necesitamos.

1. Saber cuando se instancian los vehículos, o en su defecto, cuando se activan (en caso de estar ya instanciados).

Sencillo, los objetos globales suelen ser instanciados al comienzo, y efectivamente, así es. Sólo necesitamos ver el código e insertar el envio de mensajes del tipo *Create*, para decirle al servidor que instancie los módulos correspondientes.

En este caso, lo que nos interesa se encuentra dentro del método *Awake()*. Este código se ejecutará antes de comenzar la simulación. Usando el siguiente código solucionamos este punto:

```
1 void Awake()
2 {
3   ...
4   foreach (GameObject car in cars) {
5     ...
6     // Creates a vehicle in omnet++
7     CommunicationSupport.sendCreateCar(car);
8   }
9 }
```

2. Conocer, de forma periódica, el tiempo interno de *Unity*.

Para una correcta sincronización cliente-servidor, necesitamos saber de forma periódica el tiempo del cliente, que será con el que el servidor se va a sincronizar.

Para hacer esto, vamos a emplear una variable auxiliar que utilizaremos para contar los frames que pasan, es decir, las veces que se llama al método *Update()*. Una vez que este contador llegue a un número que a nosotros nos parezca acorde a nuestras necesidades y requisitos, ejecutaremos el código correspondiente, que será el que envía el tiempo interno del cliente, y reseteamos el contador.

Sin embargo, si sólo hacemos eso, ya empezaremos la simulación con cierto retraso. Esto es debido a que el primer mensaje del tipo *ExternalTime* se enviará una vez transcurridos un cierto número de frames. Entonces, ¿qué pasa en el servidor desde que se inicia la simulación hasta que se envía ese primer mensaje del tipo *ExternalTime*? Simplemente, permanece bloqueado, ya que para él no hay tiempo máximo hasta el que poder avanzar.

¿Solución? Sencillo, enviar un primer mensaje del tipo *ExternalTime* una vez que la simulación en el cliente haya comenzado. Para ello, basta con modificar el código existente en el método *Start()* de la siguiente forma:

```
1  void Start()
2  {
3      ...
4
5      // Game start! Send first time message
6      CommunicationSupport.sendTime();
7  }
```

En cuanto a la actualización del tiempo en el servidor pasados un cierto número de frames, este es el código que lo implementa:

```
1  public class StartGame : MonoBehaviour
2  {
3      ...
4      byte frameCounter;
5      ...
6
7      void Update()
8      {
9          ...
10         frameCounter++;
11
12         // Each 10 frames
13         if(frameCounter > 10)
```

```
14        {
15          // Update server time!
16          CommunicationSupport.sendTime();
17
18          // Reset frameCounter
19          frameCounter = 0;
20        }
21      }
22      ...
23  }
```

3. Conocer, de forma periódica, la posición de los vehículos.

Para hacer esto, como en el caso anterior, nos basta con tener una variable auxiliar que emplearemos para contar los frames que pasan. Una vez que lleguemos al número de frames deseados, enviaremos la posición de cada coche.

El siguiente código implica una modificación en el código del punto anterior, por lo que a continuación se muestra como queda tras su modificación e inclusión de este punto:

```
1   public class StartGame : MonoBehaviour
2   {
3     ...
4     byte frameCounter;
5     ...
6
7     void Update()
8     {
9       ...
10      frameCounter++;
11
12      // Each 10 frames
13      if(frameCounter > 10)
14      {
15        // Update server time!
16        CommunicationSupport.sendTime();
17
18        // Send cars positions
19        foreach(GameObject _car in cars)
20          CommunicationSupport.sendCarPosition(_car);
21
```

```
22          // Reset frameCounter
23          frameCounter = 0;
24        }
25      }
26      ...
27  }
```

3.5. OMNeT++

El simulador de redes *OMNeT++* va a hacer el papel de servidor. Será el encargado de recibir y procesar los mensajes enviados por el cliente.

3.5.1. Inicialización

El primer paso para construir el servidor será crear un nuevo proyecto a través del IDE de *OMNeT++*.

Al ejecutar por primera vez el IDE, debemos indicar una carpeta para que sea el *Workspace*[6] del proyecto. Es importante que la ruta de esa carpeta **no contenga espacios**, para evitar futuros posibles problemas.

Al iniciarse, veremos un entorno muy similar a Eclipse. Sin embargo, este editor cuenta con un entorno adaptado a las necesidades denominado *Perspectiva OMNeT++*. Para aplicar esto, vamos a:

Window > Open Perspective > Simulation

En lugar de *Simulation* es posible que se llame *OMNeT++*.

Lo siguiente será crear nuestro proyecto, para ello vamos al wizard (o asistente):

File > New > OMNeT++ Project...

En el wizard solo se nos pedirá el nombre del proyecto. Además, nos da la opción de poder crear un proyecto en blanco, en blanco con una serie de carpetas, o tomando como base otras plantillas o ejemplos. En nuestro caso, optaremos por

[6]Workspace (espacio de trabajo) es donde van a estar todos los ficheros (fuente y recursos) de un proyecto en cuestión.

un proyecto en blanco, sin las carpetas por defecto.

Para concluir, tener en cuenta que los módulos en *OMNeT++* se componen de tres ficheros: fuente (.cc), cabecera (.h) y descriptor de red (.ned). Estos ficheros se pueden crear a mano, pero es recomendable crearlos a través del wizard correspondiente, es decir, mediante:

File > New > Simple Module

Esto generará los ficheros en cuestión listos para ser usados, sin riesgo de problemas en su compilación (.cc y .h) ó simulación (.ned).

Tras finalizar este proceso, ya tenemos nuestro proyecto listo para empezar a construir el simulador.

3.5.1.1. Configuración de Boost C++

Una vez compilado *Boost C++* en nuestra plataforma, tenemos que incluirlo en nuestro proyecto de *OMNeT++* para poder utilizarlo.

Para ello, en el IDE de *OMNeT++*, vamos a:

Project > Properties > C/C++ General > Paths and Symbols

Y añadimos, tanto en la pestaña *Includes* como en *Library Paths* las correspondientes rutas. En nuestro caso:

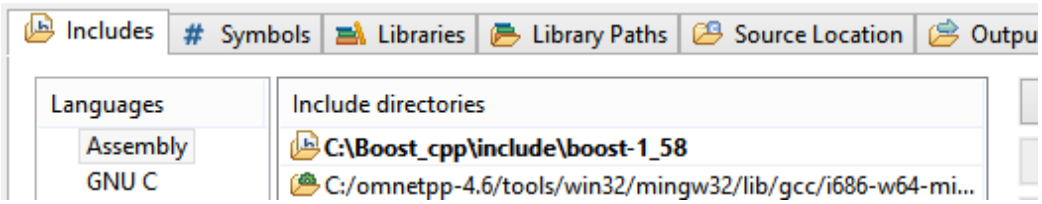


Figura 4: Configuración de *Boost C++* en *OMNeT++* (1)

Includes: C:\Boost_cpp\include\boost-1_58

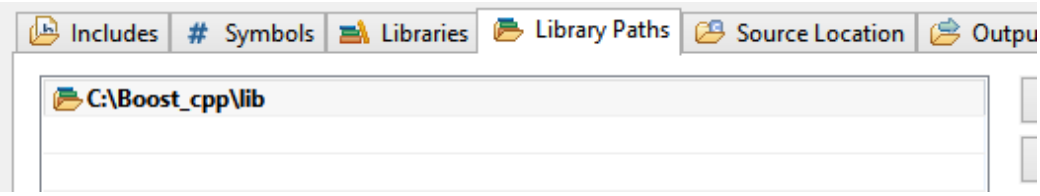


Figura 5: Configuración de *Boost C++* en *OMNeT++* (2)

Library Paths: C:\Boost_cpp\lib

Tras realizar este paso, ya tenemos *Boost C++* casi configurado. Si ahora probamos a compilar un código que haga uso de *Boost.Asio*, nos saldrá un error de referencias en las librerías.

Para solucionar esto, debemos incluir dos librerías en el makefile de nuestro proyecto. Para hacer esto en el IDE de *OMNeT++* vamos a:

Project > Properties > OMNeT++ > Makemake

Seleccionamos el makemake de nuestro proyecto y pulsamos sobre *Options...*

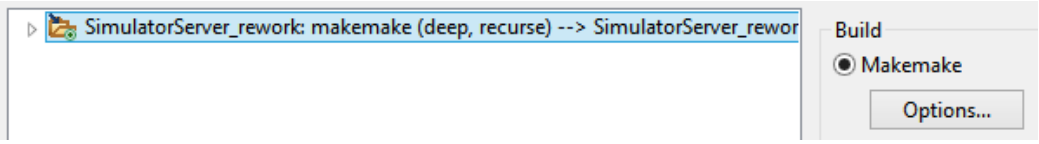


Figura 6: Configuración de *Boost C++* en *OMNeT++* (3)

Vamos a la pestaña *Link* y añadimos las dos librerías necesarias:

- boost_system-mgw47-mt-d-1_58

  El nombre de esta librería puede variar. Para saber el nombre correcto, ir a la carpeta *lib* de *Boost C++* que hemos incluido antes en nuestro proyecto y buscar la librería system para debug (contiene '-d-' en el nombre).

- ws2_32

  Librería de winsock2. Soluciona problemas a llamadas de funciones WSA en Windows.

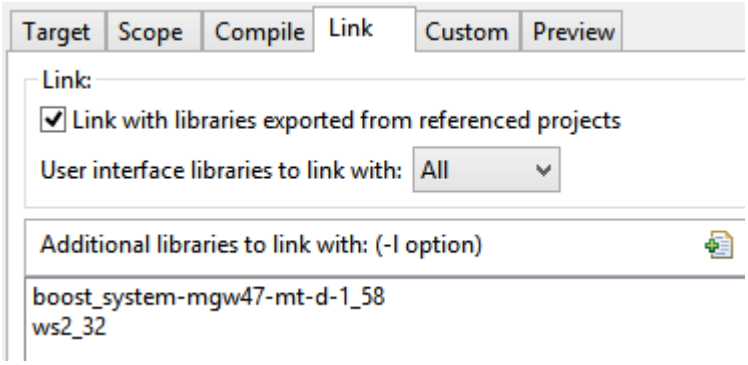


Figura 7: Configuración de *Boost C++* en *OMNeT++* (4)

Una vez completados todos los pasos, ya podemos utilizar la librería *Boost C++* en los ficheros donde se necesite. Un ejemplo, usando la librería *Boost.Asio*:

```
1 #include <boost/asio.hpp>
```

3.5.1.2. Configuración de FlatBuffers

Al igual que en el caso anterior, para poder utilizar los mensajes entre cliente y servidor debemos incluir la librería *FlatBuffers* en nuestro proyecto.

Para ello, en el IDE de *OMNeT++*, vamos a:

Project > Properties > C/C++ General > Paths and Symbols

Y añadimos en la pestaña *Includes* su correspondiente ruta. En nuestro caso:

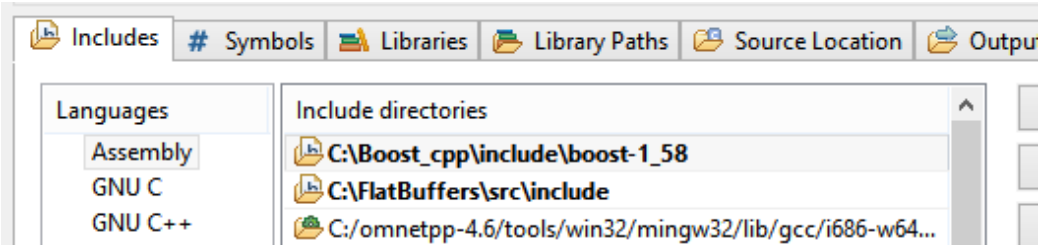


Figura 8: Configuración de *FlatBuffers* en *OMNeT++* (1)

Includes: C:\FlatBuffers\src\include

Además, deberemos incluir los ficheros generados por *FlatBuffers*. Estos ficheros contienen los mensajes que hemos diseñado, en este caso, adaptados para ser utilizados en C++.

Hemos creado la carpeta *inc_flatbuffer* en nuestro proyecto, la cual contendrá los ficheros (.h) generados. Basta con seleccionar dichos ficheros en la carpeta que han sido generados y arrastrarlos a la carpeta de nuestro proyecto.

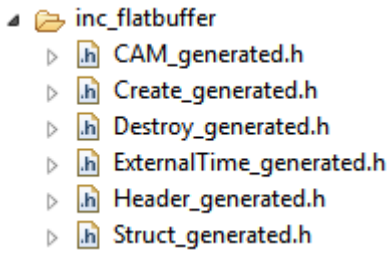


Figura 9: Árbol de cabeceras de *FlatBuffers* necesarios

Una vez hecho, ya podemos utilizar la librería *FlatBuffers* y los mensajes en los ficheros donde se necesite. Un ejemplo, en caso de necesitar el tipo de mensaje *Header*:

```
1 #include "flatbuffers/flatbuffers.h"
2 #include "inc_flatbuffer/Header_generated.h"
```

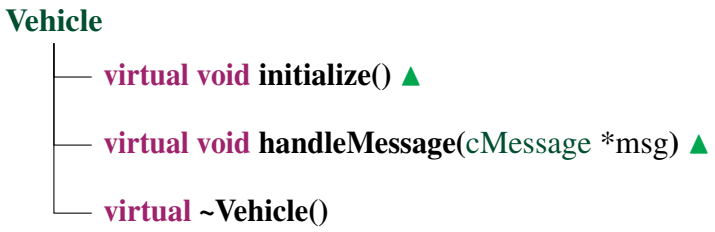
3.5.2. Clase Vehicle

Esta clase hace referencia al tipo de módulo que se instanciará en la simulación cuando el servidor reciba un mensaje del tipo *Create*, por parte del cliente.

Como su comportamiento va a ser la de un módulo, tenemos que definirlo como tal, para ello, añadimos este código al inicio de la clase:

```
Define_Module(Vehicle);
```

Hereda directamente de *cSimpleModule*. A su vez, es una clase sencilla que no incorpora nuevos métodos, por lo que su interfaz es bastante simple:



Estos métodos tienen la siguiente funcionalidad:

- **virtual void initialize() ▲**

  Se ejecuta al instanciar el módulo. Inicializa y encola un auto-mensaje al FES[7]

- **virtual void handleMessage(cMessage *msg) ▲**

  Se ejecuta al recibir un mensaje. Se comprueba si es para él o si es un auto-mensaje, y se actúa en consecuencia. En este caso, tras la comprobación, se encola en el FES un nuevo auto-mensaje.

- **virtual ~Vehicle()**

  Es el destructor del módulo. Se invoca cuando finaliza la simulación. En este caso, si por parte suya hay algún mensaje en el FES, lo cancela y lo borra.

En cuanto al fichero .NED correspondiente, sólo define la imagen que va a representar al módulo en el entorno gráfico y su posición inicial.

```
1 simple Vehicle
2 {
3     parameters:
4         @display("i=block/cogwheel");
5         @display("p=0,0");
6 }
```

Como se puede apreciar, es bastante simple, ya que este proyecto no se centra en la implementación de los vehículos. Esta clase se deberá ampliar tanto como funcionalidades tengan los vehículos en cuestión.

[7]FES (Future Events) es la lista de eventos futuros de *OMNeT++*.

3.5.3. Clase ExternalClockScheduler

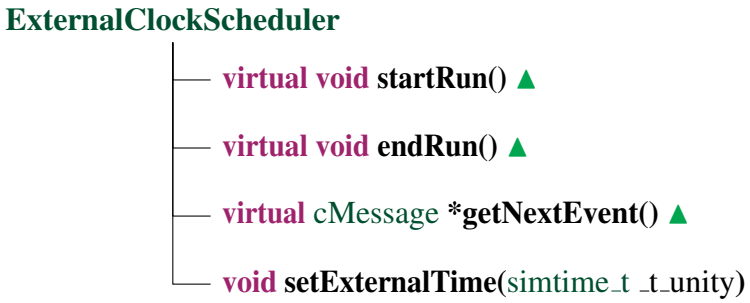
Esta clase es un planificador (ó scheduler) de eventos. Gestiona los eventos del FES. Cuando en la simulación se va a ejecutar un paso más (recordemos que funciona por eventos discretos), se llama al método *getNextEvent()* del planificador por defecto.

En nuestro caso, el planificador deriva de *cScheduler*, y se ha modificado para que devuelva los eventos únicamente si el tiempo de simulación es menor o igual que el tiempo del cliente.

Para poder usarlo, no basta con derivar de otra clase, sino que se debe registrar en la simulación. Para ello hacemos esto al inicio del código:

```
Register_Class(ExternalClockScheduler);
```

Su interfaz es mínima, se compone de tres métodos heredados y uno propio. Con sólo estos cuatro métodos podemos realizar todo lo descrito anteriormente:



Estos métodos tienen la siguiente funcionalidad:

- **virtual void startRun() ▲**

  Se ejecuta al lanzar la simulación. Inicializa la variable interna que contiene el tiempo recibido del cliente a cero.

- **virtual void endRun() ▲**

  Se ejecuta al finalizar la simulación.

- **virtual cMessage *getNextEvent() ▲**

  Obtiene y devuelve el siguiente evento del FES, siempre que el tiempo en la simulación no sea mayor que el tiempo en el cliente. Concretamente,

este es el código que realiza esta tarea:

```
1  if(simTime() <= t_unity)
2    return sim->guessNextEvent();
3  else
4    return NULL;
```

- **void setExternalTime**(simtime_t _t_unity)

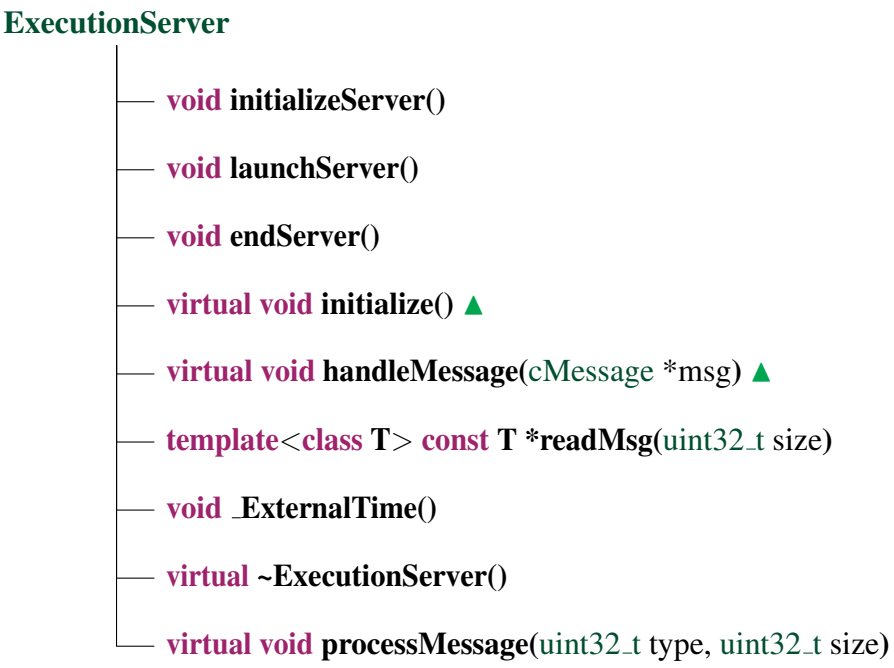
  Actualiza la variable interna con el tiempo recibido del cliente cuando el servidor reciba un mensaje del tipo *ExternalTime*.

3.5.4. Clase ExecutionServer

Es el módulo encargado de establecer la comunicación con el exterior haciendo uso de la librería *Boost C++*, concretamente, *Boost.Asio*.

Es el módulo padre, es decir, cualquier tipo de simulador que se quiera implementar, su módulo servidor deberá heredar de éste, de tal forma que todo lo relacionado con la comunicación ya esté implementado y sea transparente para el programador.

Contiene bastantes métodos, de los cuales sólo dos son heredados de *cSimpleModule*, el resto han sido implementados para el desarrollo del mismo.



Estos métodos tienen la siguiente funcionalidad:

- **void initializeServer()**

  Se invoca tras recibir el auto-mensaje. Este método es el responsable de inicializar el socket, haciendo uso de la librería *Boost*. En caso de no poder, salta una excepción.

- **void launchServer()**

  Es el servidor en sí. En su interior se encuentra un bucle infinito con flag de parada, encargado de leer los mensajes de tipo *Header* y almacenar la información que contienen.

  Además, avanza en la simulación tanto como se le permita, según el valor del tiempo del cliente.

  Para concluir, actualiza la interfaz gráfica, en caso de que estemos usándola.

- **void endServer()**

  Finaliza el servidor. Se encarga de parar el bucle infinito, cerrar la conexión e invocar al método *endSimulation()*, responsable de parar todos los módulos en funcionamiento.

- **virtual void initialize()** ▲

  Se ejecuta al instanciar el módulo. En este caso, obtendrá el puerto definido en el fichero de configuración, así como inicializar variables internas y guardar una referencia al planificador que haya por defecto.

- **virtual void handleMessage(**cMessage *msg**)** ▲

  Es el responsable de procesar los mensajes que le llegan del FES. Va a comprobar si es un auto-mensaje y, en tal caso, procede a inicializar el servidor y ejecutarlo.

- **template<class T> const T *readMsg(**uint32_t size**)**

  Es un método definido como plantilla (o template). Se ha hecho así para poder devolver cualquier tipo de mensaje que se reciba y no tener un método para cada uno de ellos.

  Su función es la de leer del socket una cantidad *size* de bytes, y devolver el objeto reconstruido según la clase que le indiquemos, usando el método *GetRoot* de *FlatBuffers*.

  Más claro, lo que hace es recibir la cadena de bytes desde el exterior y construir un objecto en base a lo recibido y a la clase que le indicamos.

  Un ejemplo de cómo se utiliza en caso de recibir un mensaje de tipo *ExternalTime*:

```
1 auto msg = readMsg<Simulator::ExternalTime>(msg_size);
2 msg->time();
```

  El método *time()* existe siempre y cuando esté definido en el que esquema que se construyo para este tipo de mensaje, tal y como se vió en el apartado de *FlatBuffers*.

- **void _ExternalTime()**

  Se encarga de recibir el mensaje de tipo *ExternalTime* y actuar en consecuencia, en este caso, actualiza el planificador con el tiempo recibido del cliente.

- **virtual ~ExecutionServer()**

  Es el destructor del módulo. Se invoca cuando finaliza la simulación. En este caso, si por parte suya hay algún mensaje en el FES, éste es cancelado y borrado.

- **virtual void processMessage**(uint32_t type, uint32_t size)

  Este método se encarga de procesar los mensajes que recibamos del exterior. El servidor se encargará de invocar esta función, indicando el tipo de mensaje que viene a continuación, y su longitud. En este caso, sólo se procesará el mensaje de tipo *ExternalTime*.

En cuanto al fichero .NED correspondiente, éste define como parámetro el puerto a utilizar, dándole un valor por defecto. Esto es útil para poder cambiar el puerto a utilizar sin tener que recompilar el código. Además, define la imagen que va a representar al módulo en el entorno gráfico.

```
1 simple Vehicle
2 {
3     parameters:
4         int port = default(9993);
5         @display("p=0,0");
6 }
```

3.5.5. Clase VenerisServer

Éste módulo se encarga de implementar la funcionalidad de nuestro simulador *Vehicular Network Realistic Physics Simulation*, también conocido como *Veneris*.

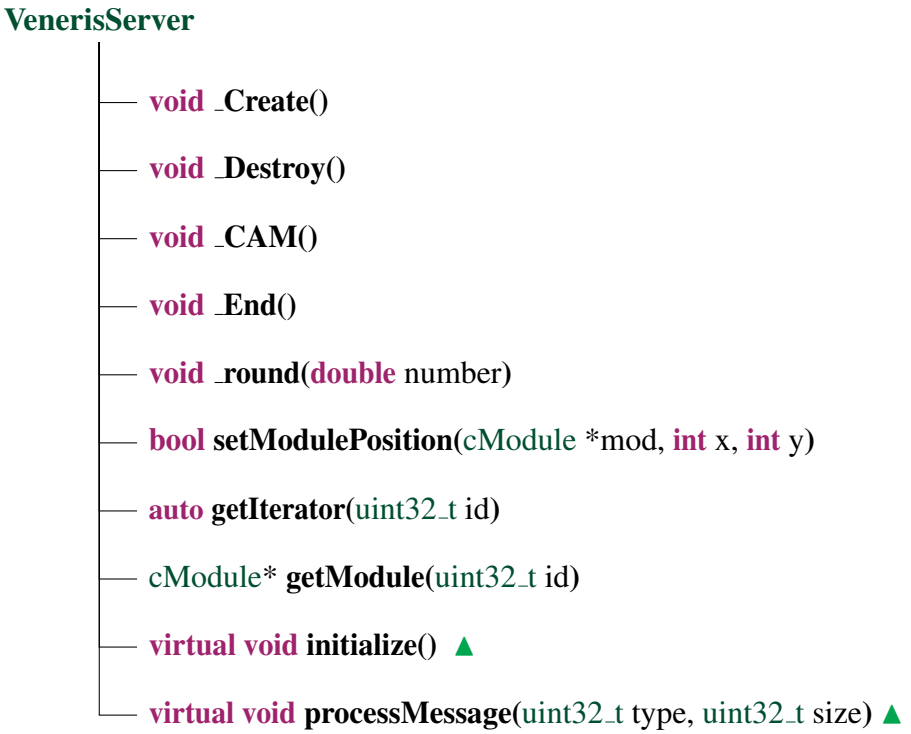
Hereda directamente de gestor de la conexión, es decir, de *ExecutionServer*. A diferencia del anterior, este si que se va a instanciar como módulo, por lo que hay que definirlo en la simulación:

```
Define_Module(VenerisServer);
```

Se va a encargar de procesar el resto de mensajes que nos llega del cliente que no sean de tipo *ExternalTime*.

Su interfaz es un poco amplia ya que implementa varios métodos para gestionar los mensajes para este tipo de simulador, aunque en general es bastante sencillo e intuitivo.

Dichos métodos son los siguientes:



Estos métodos tienen la siguiente funcionalidad:

- **void _Create()**

  Se encarga de recibir el mensaje de tipo *Create*. Una vez recibido, instancia el módulo correspondiente a ese vehículo, así como posicionarlo según las coordenadas recibidas.

  A su vez, se guarda en un mapa desordenado el par identificador del módulo y la referencia al mismo, para un rápido acceso posterior. Para usar este mapa debemos incluir la librería *Boost.Unordered_map*.

- **void _Destroy()**

  Se encarga de recibir el mensaje de tipo *Destroy*. Borra un módulo según el identificador recibido, así como borrar la entrada en el mapa correspondiente a ese módulo.

- **void _CAM()**

  Se encarga de recibir el mensaje de tipo *CAM*. Actualiza la posición de un módulo (llamando al método *setModulePosition()*, según el identificador recibido.

- **void _End()**

  Se encarga de recibir el mensaje de tipo *End*. Llama al método *endServer()* de *ExecutionServer*;

- **void _round(double** number**)**

  La posición que se recibe es del tipo double, sin embargo, los parámetros de *OMNeT++* en relación a la posición en la interfaz gráfica deben ser números enteros. Éste método se encarga de redondear adecuadamente los valores decimales, devolviendo el número entero correspondiente.

- **bool setModulePosition(**cModule *mod, **int** x, **int** y**)**

  Se encarga de actualizar la posición del módulo dado según la posición (X,Y) indicada. Devolverá *true* en caso de que el módulo exista y se produzca el cambio y, en cualquier otro caso, *false*.

- **auto getIterator(**uint32_t id**)**

  Realmente, lo que devuelve es un iterador del tipo boost::unordered::unordered_map< uint32_t,cModule*>::iterator, pero por simplicidad y ya que *C++11* nos lo permite, usamos la palabra reservada *auto* que *auto*máticamente, en tiempo de compilación, detecta el tipo de dato que corresponde y lo asigna.

  Éste iterador es respecto al mapa desordenado, cuya referencia se encuentra apuntando al par correspondiente según el identificador indicado.

- cModule* **getModule(**uint32_t id**)**

  Devuelve la referencia al módulo según el identificador indicado. En caso de no existir en el mapa desordenado, devolverá *NULL*.

- **virtual void initialize()** ▲

  Se ejecuta al instanciar el módulo. Invoca primero el método del padre, para inicializar sus variables internas y, posteriormente, se inicializan las variables correspondientes al simulador en cuestión.

- **virtual void processMessage**(uint32_t type, uint32_t size) ▲

  Este método se encarga de procesar los mensajes que recibamos del exterior. Primero se invoca el método del padre, por si el mensaje recibido es de tipo *ExternalTime*, en caso de que no, procederá a comprobar si es uno de los implementados para éste servidor, es decir, mensajes de tipo: *Create*, *Destroy*, *CAM* y *End*.

En cuanto al fichero .NED correspondiente, éste hereda de *ExecutionServer.ned*. Ya que éste tipo de ficheros permite la herencia, la usamos para evitar definir más de una vez un parámetro.

En este caso, se encarga de definir como parámetro el tipo de módulo que se va a instanciar al recibir un mensaje de tipo *Create*. Se le asignará valor en el fichero de configuración.

```
1 simple VenerisServer extends ExecutionServer
2 {
3   parameters:
4     @class(VenerisServer);
5     string moduleType;
6 }
```

Al heredar, incluye también los parámetros del padre de forma implícita.

3.5.6. Configuración de la red

Podemos definir una topología que describa visualmente la forma o estructura que tiene nuestra red.

En este caso, la red va a cambiar de forma dinámica, ya que dependerá del número de vehículos (representados como nodos en la red) y su posición. Por tanto, tendremos que definir la red base, la red inicial que debe existir como mínimo. Se define como módulo de la red el correspondiente al servidor del simulador, es decir, el módulo *VenerisServer*.

```
1 network Network
2 {
3   submodules:
```

```
4       server: VenerisServer:
5   }
```

De esta forma, al lanzar la simulación tendremos un sólo módulo principal, que es *server* y, dentro de éste, tendremos los submódulos que generemos dinámicamente[4].

Dispone, además, de un fichero de configuración, donde poder establecer valores de parámetros globales, definir el nombre de la red, etc.

```
1  [General]
2  network = Network
3  *.server.moduleType = "SimulatorServer.Vehicle"
4  scheduler-class = "ExternalClockScheduler"
```

En este fichero, se ha definido:

- El nombre de la red, denominada como *Network*.
- El tipo de módulo que se va a generar de forma dinámica. Se ha de indicar la ruta de la clase, en nuestro caso, el proyecto tiene el nombre de *SimulatorServer*, por lo que la ruta hasta nuestra clase que define el módulo es *SimulatorServer.Vehicle*.
- El planificador por defecto. Indicamos el nombre de la clase de nuestro planificador.

4. Resultados obtenidos

Tras la implementación de lo mencionado anteriormente, es hora de lanzar la simulación y ver el resultado.

Al lanzar la simulación en *OMNeT++*, nos encontramos con esta ventana:

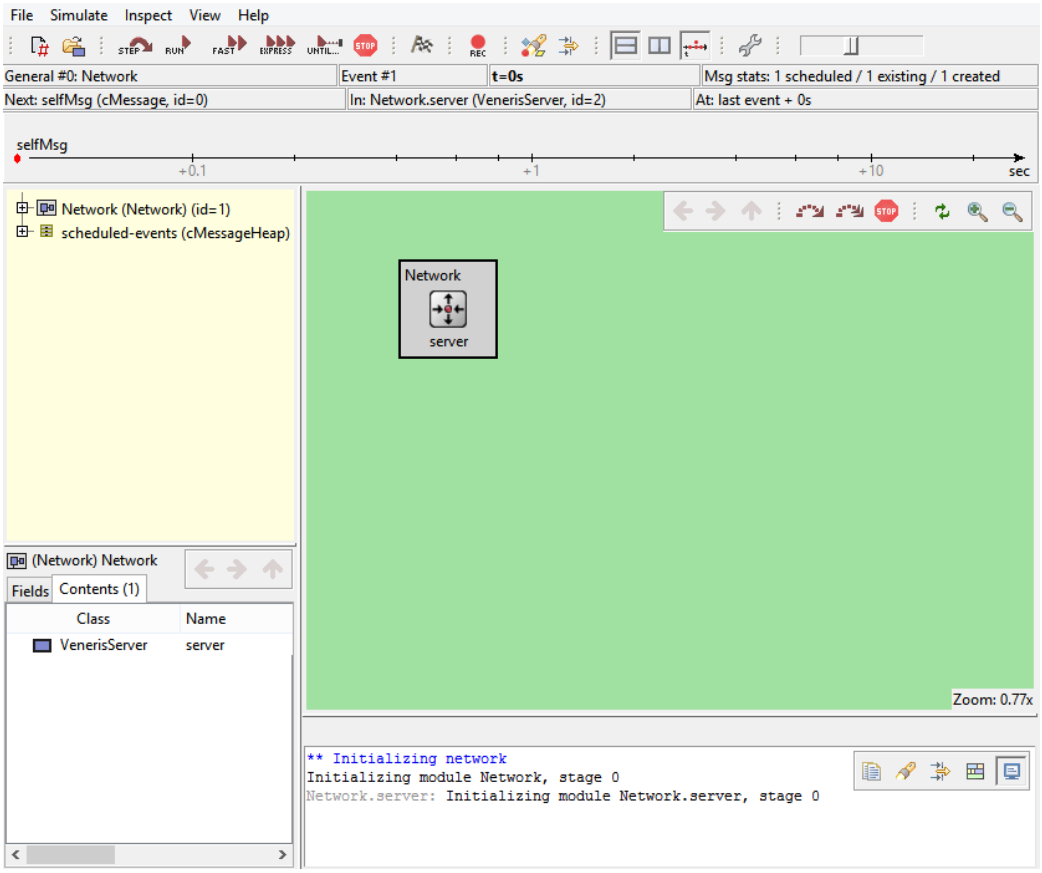


Figura 10: Interfaz de simulación en *OMNeT++*

Donde se puede apreciar el módulo principal *server*, dentro de la red *Network* que definimos anteriormente.

Tras dar al botón *Run*, se iniciará la simulación, eso provoca que quede esperando una conexión entrante por parte del cliente.

Lanzamos la simulación en *Unity*:

Figura 11: Simulación en *Unity*

Éste es el diseño gráfico que nos proporciona el paquete *UnityCar*.

Una vez iniciado, se envían los mensajes de tipo *Create* (uno por vehículo instanciado) y *ExternalTime* al servidor.

En el servidor, se generan los módulos en las posiciones indicadas. A partir de este momento, dichos módulos cambiarán su posición conforme varíe la posición de los vehículos en *Unity*.

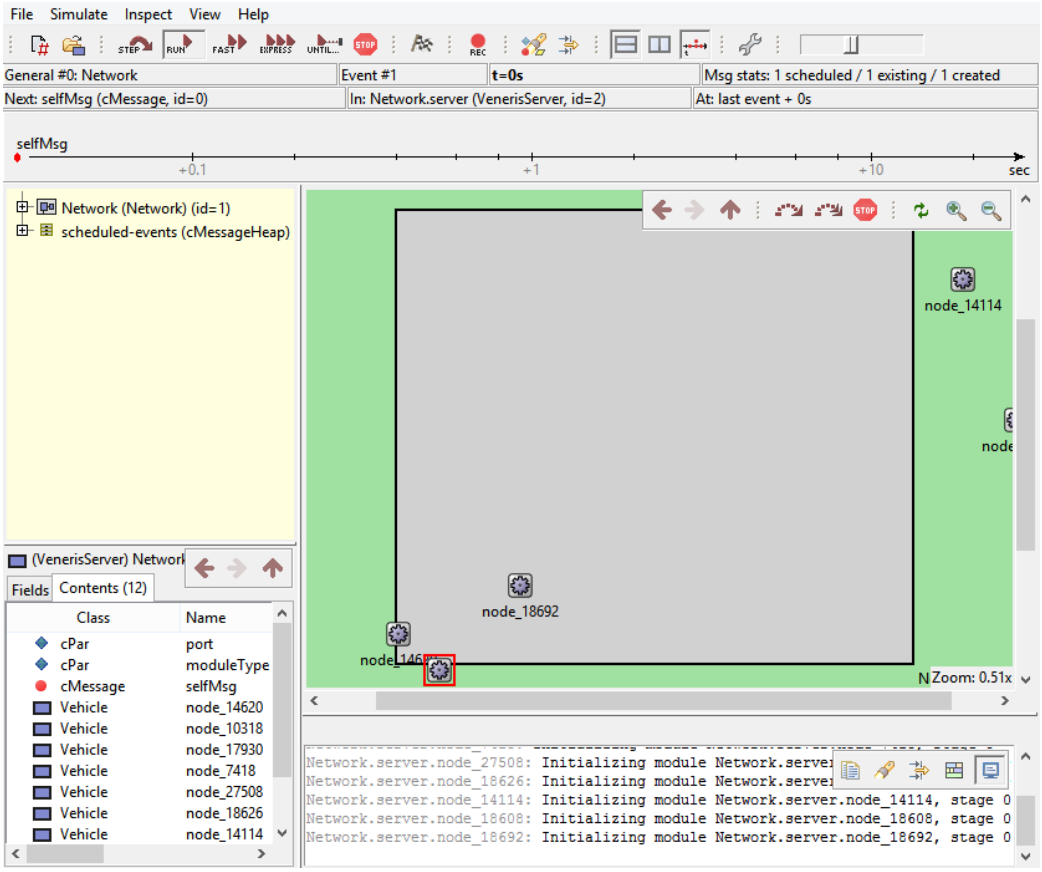


Figura 12: Simulación en *OMNeT++*

5. Conclusiones y líneas futuras

El principal objetivo de este proyecto es sentar las bases para poder realizar un simulador para situaciones de riesgo mediante la integración de un simulador de redes y un motor de videojuegos.

El problema consiste en establecer con éxito una comunicación TCP constante entre ambos simuladores, teniendo en cuenta que utilizan lenguajes distintos. Para abordar este problema, se hace uso de la librería *FlatBuffers*.

Una vez solucionado ese problema, lo siguiente es desarrollar, en ambos simuladores, las librerías necesarias para poder comunicarse con el medio e interpretar los mensajes que se intercambian.

Con esto queda una buena base con la que partir para seguir ampliando su funcionalidad, pudiendo, por ejemplo:

- Implementar una librería de funciones de almacenamiento de los datos generados por el simulador de tráfico.
- Desarrollar un prototipo demostrativo de aplicación de seguridad cooperativa sobre el simulador integrado.
- Etcétera.

6. Bibliografía

[1] A. F. Arroyo, "Implementación de situaciones de riesgo en un simulador de accidentes de tráfico mediante motor de videojuegos unity," 2015.

[2] R. H. Creighton, "Unity 4.x game development by example beginner's guide," *December 26*, 2013.

[3] C. Garcia-Costa, E. Egea-Lopez, and J. Garcia-Haro, "A stochastic model for design and evaluation of chain collision avoidance applications," no. 30, pp. 126–142, 2013.

[4] O. Ltd, "Omnet++ user manual," 2015. [Online]. Available: https://omnetpp.org/doc/omnetpp/manual/usman.html

[5] J. W. Murray, "C# game programming cookbook for unity 3d," *June 24*, 2014.

[6] U. Technologies, "Unity manual," 2015. [Online]. Available: http://docs.unity3d.com/Manual/index.html

[7] T. T. Ventures, "Oss driving simulator simwiki," 2015. [Online]. Available: http://www.transportationtechnologyventures.com/simwiki/index.php?title=Main_Page