UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Desarrollo de un videojuego para la configuración y análisis de redes de computadores

GNS3sharp

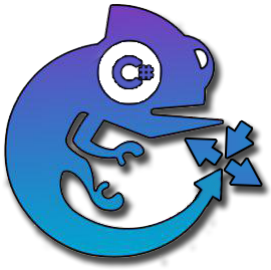
Autor
Ángel Oreste Rodríguez Romero

Directores
Juan José Ramos Muñoz
Jonathan Prados Garzón

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, agosto de 2018



Desarrollo de un videojuego para la configuración y análisis de redes de computadores

GNS3sharp

Autor
Ángel Oreste Rodríguez Romero

Directores
Juan José Ramos Muñoz
Jonathan Prados Garzón

Desarrollo de un videojuego para la configuración y análisis de redes de computadores: GNS3sharp

Ángel Oreste Rodríguez Romero

Palabras clave: palabra_clave1, palabra_clave2, palabra_clave3, ...

Resumen

El jugar ha sido desde siempre un gran amigo de la educación. Aprender jugando es un lema que cada vez se repite más. Los videojuegos, concretamente, toman en cierta forma el relevo de los juegos tal y como tradicionalmente estos se han entendido y amplían sus posibilidades.

La era digital afecta a casi todos los ámbitos de nuestro entorno. Las redes no iban a quedar excluidas de ese avance. Así, se pueden encontrar decenas de implementaciones virtuales de redes de telecomunicaciones, permitiéndonos visualizar su funcionamiento evitando el desembolso que equivale una real.

Digitalizados ambos ámbitos, ¿por qué no unirlos? ¿Y por qué no unirlos con un propósito educacional?

El presente documento pretende realizar un acercamiento a tal propósito. Se listará una serie de tecnologías que permiten llevar esto a cabo así como el desarrollo de mi aproximación.

Development of a videogame for the configuration and analysis of computer networks: GNS3sharp

Ángel Oreste, Rodríguez Romero

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Playing has always been a great friend of education. Learning by playing is a motto that is repeated more and more. Videogames, in particular, take over from games as they have traditionally been understood and expand their possibilities.

The digital age affects almost every area of our environment. Networks would not be excluded from this development. Thus, dozens of virtual implementations of telecommunication networks can be found, allowing us to visualize them working, avoiding the disbursement that is equivalent to a real one.

Digitized both areas, why not unite them, and why not unite them for an educational purpose?

This document is intended to bring this about. A number of technologies will be listed that allow this to be done as well as the development of my approach.

Yo, **Ángel Oreste Rodríguez Romero**, alumno de la titulación Ingeniería de Tecnologías de Telecomunicación de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 25351379C, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Ángel Oreste Rodríguez Romero

Granada a 1 de septiembre de 2018.

D. **Juan José Ramos Muñoz**, Profesor del Área de Telemática del Departamento TSTC de la Universidad de Granada.

D. **Jonathan Prados Garzón**, Profesor del Área de Telemática del Departamento TSTC de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Desarrollo de un videojuego para la configuración y análisis de redes de computadores: GNS3sharp***, ha sido realizado bajo su supervisión por **Ángel Oreste Rodríguez Romero**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 1 de septiembre de 2018.

Los directores:

Juan José Ramos Muñoz **Jonathan Prados Garzón**

Agradecimientos

A Juanjo, por su increíble paciencia e inestimable ayuda, tanto en nuestras tutorías presenciales como aquellas improvisadas por Telegram. A todos esos profesores que confiaron en mí y en mis capacidades más que yo mismo en tantos momentos. A todos aquellos compañeros como Alfonso que no solo me facilitaron la vida académica con su conocimiento, sino también con su compañía. A Alberto, que aunque algo reticente de primeras, está dispuesto a echarme una mano de pedírselo. A mis compañeros de Francia, que me impulsaron a ampliar mis conocimientos. A Antonio por el logo tan genial que ha hecho. A mi grupo, porque sin él no habría tenido el ánimo suficiente durante este año para afrontar el proyecto. A todos aquellos amigos que me oyeron quejarme de mi proyecto con estoicismo. A la comunidad de StackOverflow que de tantos apuros me ha sacado.

Pero ante todo, a mis padres, pilar fundamental y soporte absoluto de toda mi vida, universitaria o no. Por la educación que me han dado, por todos aquellos caprichos que me permitieron y por velar siempre por mi salud.

Índice general

Índice de figuras

Capítulo 1

Introducción

fgxdh

1.1. Introducción

huigk

1.2. Nuestro problema

En realidad, la idea principal.. quizás sea aquí donde puedes hablar de las deficiencias de juegos parecidos, o las fortalezas de usar un emulador de red real para enseñar estos temas.

1.3. Nuestra solución

Explicar qué has hecho

1.4. Motivación

En realidad, pon Motivación, y dentro, un párrafo con la motivación personal, si quieres (no es necesario poner la motivación personal).

En "motivación"pondrás qué deficiencias hay, o qué oportunidades de mejora, que hacen que este proyecto sea necesario.

1.5. Objetivos

enumerando básicamente los que pusimos en la solicitud.

1.6. Estructura del trabajo

dfsf

Capítulo 2

Planificación

fgxdh

2.1. Descripción del problema

huigk

2.2. Estructura del trabajo

Capítulo 3

Estado del arte

El estado del arte se define como el nivel de desarrollo de un ámbito concreto, generalmente relacionado con el mundo técnico-científico.

3.1. Motores de videojuegos

Pondría una subsección por cada motor a considerar. Y en cada uno, comentar por qué es adecuado o no: precio, licencia, curva de aprendizaje, plataformas que soporta, facilidad para añadir librerías, o lenguaje que utilice que sea más o menos apropiado, si hay mecanismos para comunicarte con software externo (como el caso del emulador)... Con media página por cada uno, creoque basta. Unreal Engine, Amazon Lumberyard, Game Maker, Godot Engine, y no sé si alguno más que sea famosete, indicando que son los más populares (ojalá haya un informe con lista de usuarios por engine...).

Creo que tanto para hablar de Unity como de GNS3, lo suyo es:

comentar las características generales en "motores de juegosz en la de .emuladores". y luego añadir una sección quizás con los detalles de las plataformas elegidas, hablando de las características que vas a utilizar (API de GNS3, capacidad de enlazado con DLL de Unity...). Cosas que vaya a necesitar el revisor para entender el diseño o sobre todo la integración.

3.1.1. Unreal Engine

3.1.2. Amazon Lumberyard

3.1.3. Game Maker

3.1.4. Godot Engine

3.1.5. Unity

3.2. Simuladores de redes

3.2.1. Simulador 1

3.2.2. Simulador 2

3.2.3. Simulador 3

3.2.4. GNS3

3.3. Nuestra elección

3.3.1. Unity

3.3.2. GNS3

hay que comentar qué mecanismos nos interesan de GNS3. Si no se ha explicado cómo se organiza y funciona antes, aquí es el lugar.

GNS3 es usado ampliamente como método de entrenamiento para los exámenes de Cisco. Tal es así, que cuenta con su propia academia online de cursos en los que se utiliza el simulador para estudiar, convirtiendo la formación en más accesible. Sin embargo, las imágenes de las máquinas de Cisco, aunque pueden ser encontradas fácilmente en internet, requieren de licencia para ser usadas. Siendo así se optó por hacer uso de software libre para el proyecto.

Nodos

Una figura esquemática que muestre todos estos conceptos, es fundamental al comienzo de un capítulo o sección importante.

Cada elemento de una red está representado en GNS3 por un elemento llamado **nodo**. Estos nodos, que pueden ser desde un router a un switch,

no son más que virtualizaciones de aparatos reales. De normal, estas virtualizaciones se realizan en base a imágenes de los sistemas operativos que se integran en los aparatos. Así, podemos tener varios routers distintos de Cisco montados sobre la misma estructura, permitiéndonos jugar con ellos con verdadera facilidad.

Enlaces

La API de GNS3

Una API (que será explicada más adelante) La primera vez que aparezca un acrónimo, debes indicar cuál es su significado. De hecho, en los títulos o como primera palabra de la frase, (o en el abstract) hay que evitar las abreviaturas.

3.4. Juegos docentes

Con todo lo anterior nuestro objetivo es tal. Ya hay ejemplos

Deberían incluirse juegos que ya existan. Añade una subsección çonclusiones", o algo en la que compares qué tienen y debe incorporar tu juego, y qué cosas les falta, que vas a poner en el tuyo.

Capítulo 4

Diseño de la solución

En este capítulo se describirá el modo en el que la API será construida y cómo el juego ha de ser llevado a cabo para cumplir con los propósitos que se han ido repitiendo a lo largo del texto.

4.1. La API: GNS3sharp

Para llevar a cabo la interacción entre el videojuego y el simulador de redes es necesario desarrollar previamente alguna herramienta que lo posibilite. Con esta fin, se ha construido una librería propia que lo posibilita.

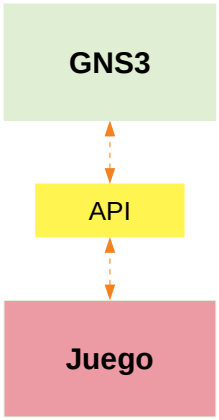


Figura 4.1: La API en el proyecto

El lugar de esta API (aquí usaremos indistintamente API y librería) en nuestro proyecto se encuentra entre el juego y GNS3 tal y como se deja entrever en la figura 4.1. Todos sus detalles serán explicados en las próximas

páginas.

4.1.1. Introducción

Según Wikipedia, una **API** "es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción"[1]. Comprende una serie de funciones que facilitan en mayor o menor medida trabajar sobre un cierto elemento. Como ejemplo de API famosa tenemos la de Google Maps, que contiene un compendio de métodos para JavaScript que permiten interactuar directamente con la plataforma de Google y crear nuestros programas jugando con ella[2]. En resumen, una API no es más que un conjunto de funciones definidas para acceder a un servicio determinado.

La programación se rige por **capas de abstracción**. Partiendo de conceptos concretos se desarrolla una capa de abstracción haciendo uso de ellos que permite elevar el trato de elementos concretos un nivel por encima[3]. De esta forma ganamos en agilidad de escritura y sencillez en el diseño sin perder flexibilidad de desarrollo.

Pondría más bien que para facilitar esta comunicación, una de las alternativas que exsiten en sistemas complejos, es aplicar el concepto de capa. Una capa sería.. (en redes, el modelo OSI o el TCP/IP usan capas, en la que la superior solicita servicios a la inferior, y esta se lo facilita... En TCP/IP se define de forma abstracta, no a nivel de programación).

Como se ha citado previamente, GNS3 hace uso de una **API REST** (*REpresentational State Transfer*). Esto es, mediante una serie de métodos (los conocidos GET, POST...) asociados a una URI concreta podemos interactuar **vía web** con una aplicación. La diferencia fundamental con otra clase de servicios web es que REST está orientado a recursos y no a métodos. Esto permite a la web utilizar comunicaciones sin estado, facilitando de este modo su escalabilidad[4].

Aunque de gran utilidad (ya veremos qué papel concreto juega en nuestro trabajo), hace falta algo más para poder propiciar la interacción entre el simulador de redes y el videojuego; la API trabaja a un nivel mucho más bajo del que requerimos para desarrollar los juegos. Surge la necesidad entonces de crear nuestra propia API que haga uso de la API REST de GNS3 y que defina las funciones necesarias, a un nivel más abstracto y con más facilidades, que haga de intermediaria entre GNS3 y Unity.

4.1.2. Elección del lenguaje

En pleno 2018 la cantidad de lenguajes de programación existentes roza el absurdo. Desde el tradicional C, pasando por el multifuncional Java, el sencillo Python o incluso los llamados lenguajes esotéricos como LOLCODE[5]. De entre todos ellos nosotros elegiremos uno sobre el que trabajar. Esta decisión está condicionada, como es natural, por el motor de videojuegos a utilizar.

Ya que nuestra intención es que el motor sea capaz de establecer interacción con el simulador, es necesario que la API a desarrollar esté escrita en un lenguaje con el que el propio motor sea capaz de trabajar. **C#** parecía la opción más sensata. ¿Por qué? Las razones se exponen a continuación:

- **Porque es el lenguaje más usado en Unity**. Unity admite varios lenguajes de programación con los que desarrollar los scripts asociados a los juegos. C++, usado en otros muchísimos otros motores de videojuegos como Unreal Engine, es uno de ellos. Aunque se trate de un lenguaje increíblemente potente y eficiente, su complejidad de uso es mucho mayor, ya que se encuentra a más bajo nivel. JavaScript es otro de ellos, pero no es tan recomendable como C#, pues entre otras razones, a diferencia de JS, C# es fuertemente tipado[6]. En general, C# es el lenguaje usado por defecto en Unity y el recomendado por todos, documentación incluida.

- **Porque son más motores quienes lo utilizan**. Unity está en el podio de entre los motores de videojuegos más usados en el mundo. Como tal se convierte en un referente. El resto de motores miran hacia él y, si quieren atraer a nuevos programadores, tendrán que hacer de su incursión en el nuevo motor algo sencillo. Este es el caso de Godot Engine, que unos meses atrás decidió incluir tal lenguaje entre los soportados[7]. Esto quiere decir que la API no solo podrá ser usada en juegos creados en Unity, si no que su terreno de juego se verá ampliado. CryEngine es otro de los motores que permiten el uso de C# como lenguaje de scripting.

- **Porque, ante todo, es un gran lenguaje**. C#, similar en cuanto a sintaxis a Java, nació como respuesta a este de la mano de Microsoft. Se trata de un lenguaje de propósito general, aunque es usado primordialmente para la construcción de aplicaciones para infraestructuras Windows. Es uno de los lenguajes que componen la plataforma .NET de Microsoft. Tal es su importancia que a día de hoy se posiciona como el cuarto lenguaje de programación más usado a nivel mundial[8]. Alguna de las características que lo hacen especialmente atractivo:

- Es un **lenguaje de programación orientado a objetos**, con lo que posee todas las características propias de estos (encapsulado, herencia y poliformismo). Sin embargo, no admite multiherencia. Su componente fundamental es una unidad de encapsulamiento de datos y funciones llamada `type` o "tipo". C# tiene un sistema de tipos unificado, donde todos los tipos en última instancia comparten un tipo de base común.
- Aunque es principalmente un lenguaje orientado a objetos, también **toma prestado del paradigma de programación funcional**. Las funciones pueden ser tratadas como valores mediante el uso de delegados, permite el uso de expresiones lambda, acercándose a los patrones declarativos del paradigma funcional...
- Admite **tipado estatico**, lo cual se traduce en que el lenguaje obliga a que haya coherencia entre los tipos durante el tiempo de compilado. El tipado estático elimina un gran número de errores antes de que se ejecute un programa. Desplaza la carga del momento de la ejecución hacia el compilador para verificar que todos los tipos en un programa encajan correctamente. Esto hace que las aplicaciones grandes sean mucho más fáciles de administrar, más predecibles y más robustas. Además, la escritura estática permite que herramientas como IntelliSense en Visual Studio ayuden a desarrollar, pues conoce el tipo de una variable determinada y, por lo tanto, qué métodos puede utilizar está habilitada a usar. C# incluye además el tipo `dynamic` que permite sortear el tipado estático y dejar que el tipo de variable se averigüe durante el momento de la ejecución[9].

Particularmente, nosotros haremos uso de C#7, lanzado a la vez que Visual Studio 2017. Entre sus características más interesantes se incluye la aparición de tuplas, especialmente útil cuando queremos devolver más de una variable como resultado de una función.

Aclarada las razones, pasamos a analizar el uso que le daremos a este lenguaje.

4.1.3. Diseño de la API

Antes de desarrollar la API, que se ha decidido llamar **GNS3sharp** por la fusión entre las dos tecnologías que la conforman, es necesario reflexionar sobre la forma que tendrá. Para dar soluciones, tenemos que plantearnos antes las preguntas adecuadas:

¿Cómo debe interactuar con el simulador?

Como ya se dijo con anterioridad, GNS3 crea un servidor en el equipo desde el que se ejecuta. Este servidor acepta peticiones REST, creando así una suerte de interacción con el programa sin necesidad de poner las manos en él de forma directa. Se abre de esta forma al mundo del scripting y así a nosotros para crear un conjunto de métodos que faciliten su acceso y gestión. Todo esto volverá a ser ilustrado en la subsección 4.2.1 con mayor detalle.

Tratándose de una API REST, lo único que nos es necesario para la conexión es un **cliente web**. El framework de .NET ya cuenta con una clase dedicada a ello denominada `System.Net.WebClient`[10], que contiene métodos más que suficientes para nuestro uso. Será pues sobre esto sobre lo que se base primordialmente la relación entre nuestra librería y el simulador de redes como tal. A continuación veremos que habremos de emplear alguna tecnología más.

¿Cómo queremos que interactúe con el exterior?

Es importante tener esto claro, pues el uso de la librería y de sus métodos por parte de una aplicación externa debe optimizar la productividad y agilizar el desarrollo lo máximo posible. Aquel que vaya a hacer uso de ellas debe tener un esquema claro del modo en que puede gestionar la interacción, siendo pues cercana al usuario (más bien desarrollador) último.

GNS3 funciona a través de proyectos. En un proyecto se pueden incluir tantos nodos como se desee para, a posteriori, ser interconectados. Las posibilidades son muy amplias y dejan poder absoluto al usuario. La singularidad del proyecto apenas pasa por permitir arrancar o apagar todos los nodos contenidos en él a la vez.

Siendo así, se ha optado por concentrar lo crucial de la interacción entre el simulador y el desarrollador en **una sola clase-objeto**. Esta clase, que se explicará con detalle más adelante, permitirá hacer de puente entre el proyecto y los distintos elementos que lo componen y aquel que haga uso de ella.

Más importante aún, ¿qué queremos que haga?

Todo lo explicado hasta ahora es correcto: definimos una forma con la que el código pueda acceder al simulador y reflexionamos en la manera en la que el desarrollador que la use pueda sacarle partido. Sin embargo, ¿qué significa entonces *sacarle partido*?

Para responder no hay más que irse al propio GNS3 y ver todas las opciones que nos ofrece: desplegar nodos, enlazarlos entre sí, cortar esos

enlaces, arrancar todos los nodos juntos, parar algunos de ellos a voluntad...

Y por supuesto y aún más importante, **gestionar los aparatos desplegados desde dentro**. Es aquí donde está su verdadero potencial. Manejar un switch conectado a varios PCs, modificar sus VLANs dinámicamente; un router conectado a varias redes distintas, desactivando y activando algunas de ellas en función de un booleano con el que estemos trabajando... Y todo esto sin necesidad de acceder a GNS3 directamente; todo mediante scripting. ¡Se abre un mundo de posibilidades!

4.1.4. Diseño de clases

Nuestra API hará uso de todas las características propias de los lenguajes orientados a objetos ya expuestas. A nivel de estructura, la que nos interesa citar ahora es la **herencia**.

La herencia es un mecanismo por el que una clase hija (llamémosla `B`) va a heredar los métodos y las propiedades de una clase padre (llamémosla `A`). La cantidad de elementos heredados entre padre e hijo puede ser determinado mediante el uso de modificadores de acceso. Como dato adicional, este lenguaje no admite herencia de constructores, así que nuestra clase `B` tendrá que definir su propio constructor (o bien explicitar su herencia respecto al de `A`).

Aclarado el concepto, mostramos la estructura básica sobre la que nuestra librería será construida:

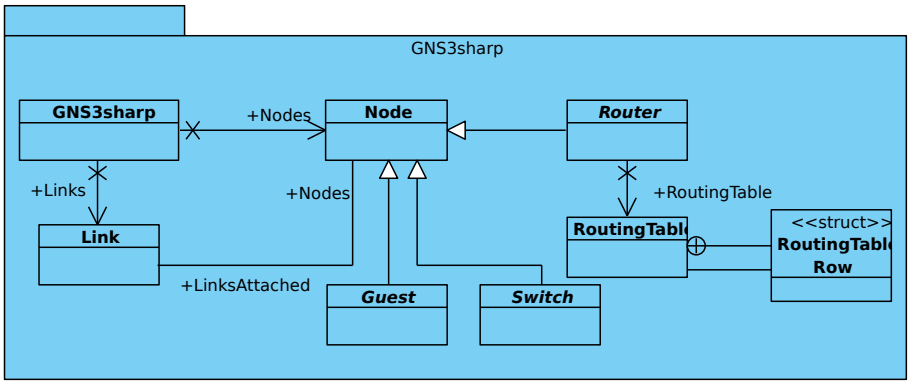


Figura 4.2: Boceto de diagrama UML de la API

El diagrama UML de la figura 4.2 dibuja cuáles son las clases que componen la librería y da pistas acerca de la relación entre ellas. Así, es claro que las tales `Switch`, `Node` y `Router` heredan de de `Node` y que a su vez esta es usada por `GNS3sharp` por alguna propiedad llamada `Nodes` (que hasta la subsección 5.1.1 no se analizará).

A continuación pasamos a desarrollar cada una de las clases conceptualmente:

GNS3sharp, la clase principal

Tal y como expusimos en la sección 4.1.3, crearemos una clase principal que gestionará toda la interacción con el simulador de redes. Hemos llamado a esta clase del mismo modo que a la API, `GNS3sharp`. Recalcamos entonces: los objetos que se encarguen de gestionar los proyectos serán del tipo `GNS3sharp`.

¿Y en qué consistirá esta gestión concretamente? Básicamente, la clase debe encargarse de:

1. Establecer la conexión con el servidor de GNS3 y recopilar toda la información acerca del proyecto que se pretende controlar.

2. Convertir toda es información en objetos útiles que puedan ser utilizados.

3. Crear una gestión eficiente de esos recursos de manera que sean fácilmente accesibles y manipulables.

Estos recursos de los que hablamos serán en gran medida los representados por las clases que se desarrollarán justo debajo.

Sin embargo, la creación de proyectos y despliegue de nodos en los mismos quedarán excluidos, ya sea por la complejidad añadida que conlleva o porque no son funciones que nos sean vitales para la construcción de juegos. Estas funcionalidades deberán ser llevadas a cabo manualmente. Más adelante podrán ser controladas desde la librería sin problema.

Node

Sin lugar a dudas es aquí donde se encuentra la característica más interesante de GNS3 y es hacia donde nuestros esfuerzos deben dirigirse. Dado que cada nodo representa un aparato de la red, lo ideal es que ese aparato pueda ser convertido a un objeto en nuestra biblioteca desde el que se nos permita su control.

Esta es la finalidad de la clase `Nodo`. Su deber será el de contener todos los parámetros necesarios para habilitar la conexión con el nodo y así abrir un canal de comunicación con él; un canal que habilite tanto el envío como la recepción de mensajes.

Para la creación de las instancias de la clase se tendrá que recurrir a `GNS3sharp`, que mediante los datos que recoja del servidor de GNS3 será capaz de crear asimismo el objeto.

Como GNS3 admite todo tipo de aparatos de red, cada uno con sus peculiaridades, lo más sensato es crear una clase para cada uno de esos equipos. Esta individualización permite definir métodos propios para cada elemento y, de este modo, facilitar su uso final. Tal y como se puede ver en el diagrama de la figura 4.2, definiremos tres clases principales:

1. `Guest`, representante de dispositivos *end-point* como son los PCs.
2. `Router`, representante de dispositivos de la tercera capa OSI.
3. `Switch`, representantes de dispositivos de la segunda capa OSI.

Todas ellas heredarán de `Node`. De estas clases nacerán asimismo otra serie de clases referidas a aparatos concretos y a no tipos genéricos.

Link

Los nodos están interconectados entre sí mediante enlaces. Representaremos cada uno de ellos mediante la clase `Link`. Esta clase guardará como objetos a los nodos que interconecta así como información sobre el propio enlace (cuánto jitter, latencia... posee). Al igual que `Link`, la generación de instancias de esta clase pasará por `GNS3sharp`.

La clave de representar los enlaces en la API es que nos permitirá añadir, eliminar o incluso eliminarlos del proyecto desde el código, ampliando las posibilidades de interacción con el simulador.

Otras clases

Además de las anteriormente expuestas, se creará otra serie de clases que, o bien ayuden al desarrollo de la API (una clase con funciones de ayuda), o bien faciliten la representación de los datos (tal será el caso de `RoutingTable`, que puede atisbarse en la figura 4.2).

4.2. Diseño del videojuego

Establecida una plataforma sobre la que GNS3 pueda interactuar con código, llega el momento de diseñar el videojuego que haga uso de ella.

4.2.1. Interacción con el emulador

A continuación se dará una visión de la interacción desde el punto de vista del motor del videojuego; desde la creación del juego. Se facilita el siguiente esquema a la espera de que sea de ayuda a la hora de comprender la sección.

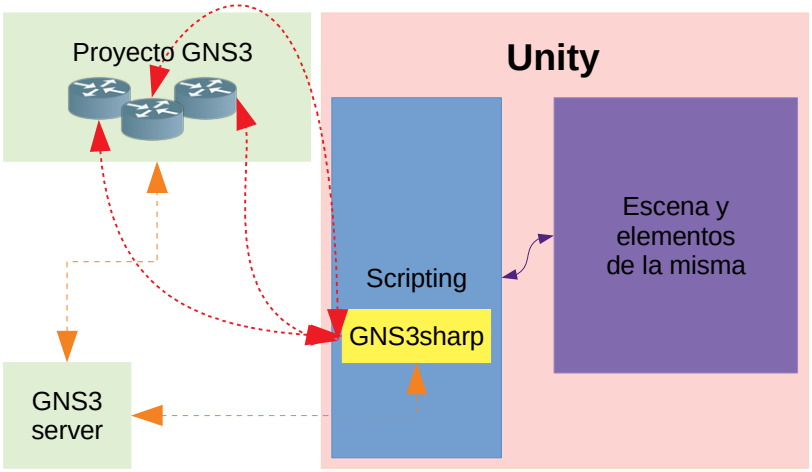


Figura 4.3: Diagrama de la interacción entre Unity y GNS3

Los proyectos de Unity están basados en un elemento llamado `GameObject`. Todos los objetos que componen las escenas del proyecto se pueden abstraer en última instancia en ese tipo. El primer paso para que un proyecto de Unity pueda conectarse y controlar un esquema de red de GNS3 pasa por **insertar un `GameObject` que tenga una instancia de la clase `GNS3sharp`**. La única condición que se le impone a este objeto es que este parámetro contenido en él (la instancia) ha de ser accesible por todo el resto de elementos de la escena.

¿Para qué? De este modo, todos esos elementos no necesitan crear la suya propia. Creando una sola instancia de la clase centralizamos el trabajo en un solo objeto. La instancia contendrá información de los datos del proyecto de GNS3; información que podrá ser tomada por todos los componentes de la escena sin más que copiar la referencia al objeto. Si estos declararán su propia instancia cada uno perdería en tiempo (la recopilación de datos desde el servidor no es instantánea) y en eficiencia.

La idea es que el jugador (en otros términos, el usuario final), posea interacción con el simulador de redes de forma que **las consecuencias puedan verse reflejadas en el juego**. Pongamos un ejemplo e invirtamos el orden lógico de desarrollo para entender mejor cómo funcionaría esto:

1. El jugador se encuentra en una habitación que no es más que una **representación de un router**. Tiene un botón justo delante suya. En la pantalla se le avisa de que si lo pulsa, la puerta asociada a una determinada interfaz se cerrará, de forma que los enemigos que se ven acercarse en esa dirección no podrán pasar. Lo pulsa entonces.

2. Ese botón es un `GameObject` de la escena actual que tiene un *trigger* asociado a él y un script que lo gestiona. Uno de los atributos de tal código es la referencia a la instancia del objeto `GNS3sharp`, construido al arrancar la escena y que suponemos ya contiene toda la información del proyecto de GNS3. Cuando se activa el trigger (una condición del script pasa a ser `True`), se toma la referencia del objeto asociado al router donde el personaje está. Se utiliza tal objeto para enviar un comando al router que le haga apagar la interfaz correspondiente.

3. El mensaje llega al router. Este lo ejecuta y devuelve el resultado.

4. El script recibe la respuesta y, tras analizarla, confirma que la interfaz se ha cerrado. Envía un mensaje al `GameObject` de la puerta para que la cierre.

Este flujo es de alguna forma lo que se ha representado en la figura 4.3. El bloque morado hace referencia a todo el entorno que comprende una escena de Unity. Esta escena contendrá una serie de `GameObject`s, cuyo comportamiento vendrá determinado mayormente mediante lógica programada por scripts. El bloque azul hace referencia a este compendio. Alguno de esos scripts será el encargado de contener una instancia de `GNS3sharp`. Al ser creada, esta se conecta al servidor de GNS3 desde donde recibe toda la información del proyecto con el que se quiere trabajar. A partir de ahí, el resto de `GameObject`s de la escena no tendrán más que usar esa instancia para poder conectarse a los nodos (entre otras cosas) de la simulación.

4.2.2. Modelo de videojuego

Ahora que sabemos de qué forma se interconecta el juego con la red y en qué consiste exactamente la interacción, queda por concretizar qué clase de juegos podríamos crear. Hay que tener en mente en todo momento que pretendemos crear un **juego didáctico**, así que no nos vale cualquier construcción. La dificultad es, entonces, doble, pues es necesario seguir los cánones que dictan el buen hacer de un videojuego (véase un inteligente diseño de niveles, una estética atractiva y coherente, apartado técnico gráfico y sonoro propio, niveles a prueba de bugs...) a la vez que planear **qué** se quiere enseñar en ellos y **cómo** se pretende hacerlo.

Para sortear estas dificultades se ha decidido que la estructura del juego esté basada en pequeños niveles interconectados entre sí por una plataforma común. La linealidad de un juego exige de un guion y de una cohesividad de los que el juego "federado" puede permitirse prescindir.

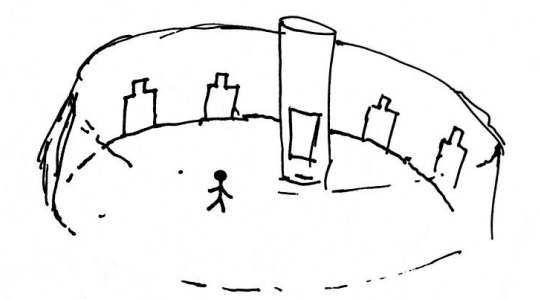


Figura 4.4: Boceto del juego

En el boceto de la figura de arriba se quiere ejemplificar (de forma extremadamente simplista) en qué consistiría este diseño.

- Cada vez que el jugador entre en el juego se encontrará en una suerte de "hall". El hall estará acotado por paredes con un número determinado de puertas.

- Estas puertas conducen a los distintos niveles del juego. Cada puerta corresponde a una lección diferente sobre redes y telemática. El jugador no tiene que superar una lección (un nivel, una zona) para acceder a la siguiente, sino que puede elegir qué quiere aprender indistintamente. Por supuesto, se podría vetar la entrada a una zona hasta que supere una concreta. Idealmente, cada nivel al que se accede contaría con varias fases que van incrementando en dificultad.

- Al superar estas zonas, se desbloquea un "manual" acerca de lo aprendido en esa sala. Este manual puede ser consultado en un monitor localizado en el centro del hall. Gracias a esto, se podría repasar rápidamente lo aprendido en niveles anteriores desde un solo lugar.

Esta forma de acceder a los niveles es similar a la que siguen los videojuegos de la saga *Crash Bandicoot* (*2*, *3*, *La venganza de Cortex...*), donde existe una sala de "descanso" desde la que elegir cuál será la siguiente fase a jugar. Abstrayendo esta idea algo más, no es más que un menú estático desde donde elegir el nivel (juegos de corte más "clásico" como *Super Meat Boy* utilizan este patrón) salvo que gozando de una interacción más directa y natural con el jugador, integrándolo más.

Existen dos aproximaciones al modo en el que estos niveles se conectan con el simulador de redes:

- Por un lado, podría existir **un proyecto de GNS3 por cada nivel** y/o fase. Aunque mucho más estructurado y por ende posiblemente más sencillo de gestionar, es necesario establecer conexión con el servidor de GNS3 por cada nivel para recopilar la información del mismo. Esto se traduce en tiempo. Además, el servidor de GNS3 no registra el proyecto de GNS3 a menos que este se abra manualmente desde GNS3 en una sesión. Más tiempo.

- La otra solución pasa por crear **un solo proyecto** para todo el juego. Sería necesario desplegar todos los aparatos que vayan a ser utilizados en él, incrementando altamente los recursos que el sistema utiliza. Para paliar este último problema existen dos opciones:

  1. Encender los aparatos que vayan a ser necesitados en el nivel a jugar y cerciorarse de que el resto permanece apagado.
  2. Reconfigurar los aparatos que vayan a ser utilizados para que sus propiedades se correspondan a lo que se necesita para el respectivo nivel.

  De entre estos dos acercamientos al problema, quizás el primero sea más simple, pero el arranque de cada nodo no es inmmediato. El segundo entraña una mayor dificultad ingenieril y mayor propensión al error aunque la configuración de los aparatos conlleve menos tiempo.

Sin embargo, aunque todas estas ideas pueden sonar realmente interesantes, el juego final desarrollado no ha podido alcanzarlas. En el siguiente capítulo, donde se detalla la integración real de lo expuesto en este capítulo, se comprobarán cuáles han sido las dificultas encontradas en el camino que han frenado de una forma u otra el escenario inicial.

Capítulo 5

Integración

Trazado el esquema del trabajo a realizar, el siguiente paso es lógicamente llevarlo a cabo. Las soluciones empleadas para todo el proceso será lo que compongan este capítulo.

5.1. Desarrollo de la API

Si bien en la sección 4.1 ya se habló de la estructura con la que nuestra API contaría, en esta ocasión se contará con todo detalle el modo en el que esta ha sido desarrollada. Se describirán los puntos más importantes de cada una de las clases así como el modo en que fueron originados. Naturalmente, no se pretende explicar todo el contenido, pues no es su propósito. Se evitará así explicar en muchos casos métodos y propiedades privadas para centrarnos en aquellas públicas.

Para finalizar se explicará la relación existente entre las clases con ayuda de un diagrama UML así como la forma en la que estas han sido compiladas para ser unificadas como librería.

Todas estas clases conforman un único `namespace` o espacio de nombres. Conviene recalcar que la API ha sido enteramente desarrollada desde cero.

5.1.1. GNS3sharp

Constructor

El constructor de `GNS3sharp` es sin duda uno de los elementos más importantes de toda a API. ¿Por qué? Tendrá la responsabilidad de conectarse al servidor de GNS3, recibir todos los datos de un cierto proyecto que solicitemos, procesarlos y guardarlos de tal forma que sean útiles. Su cabecera

se muestra justo a continuación:

Código 5.1: Cabecera del constructor de `GNS3sharp`

```
public GNS3sharp(string _projectID, string _host = "
    localhost", ushort _port = 3080)
```

Cosas que serán necesarias entonces para inicializarlo:

- **ID del proyecto**: cada proyecto de GNS3 tiene un ID asociado que el servidor guarda junto a su referencia. Más adelante, en la subsección 5.1.5, veremos cómo no es necesario conocer el ID del proyecto sino que con el nombre del mismo será más que suficiente.
- **Host**: la dirección del equipo donde el servidor está alojado. Por defecto se toma `localhost`; se supone que la mayor parte de las veces se encontrará en el mismo equipo desde el que se utiliza la API.
- **Puerto**: además de la dirección del servidor, es necesario conocer el puerto en el que está montado. GNS3 determina el `3080` por defecto.

De acuerdo, ¿y qué hace exactamente con estos tres parámetros?

1. Crea la cadena de texto de la URI donde está el recurso asociado a los nodos del proyecto. En esta dirección existe únicamente un JSON con toda la información sobre él. Se instancia un cliente web y **se descarga el recurso como cadena de texto**.
2. Hay que **deserializar el JSON**. Este paso no es en absoluto trivial, ya que los métodos de *Json.NET* son incapaces de extraer los datos que necesitamos directamente. Así que, valiéndonos de las herramientas en forma de clases que nos ofrece, lo hacemos manualmente.

Código 5.2: Deserialización de JSON

```
// JSON array object
JArray jsonArray = JArray.Parse(json);
Dictionary<string,object> tempDict = new Dictionary<
    string, object>();

// Variables in which store the JSON info temporaly
string name; object value;
if (jsonArray.HasValues){
    foreach (JObject jO in jsonArray.Children<JObject
        >()) {
        foreach (JProperty jP in jO.Properties()) {
            name = jP.Name;
            value = (object)jP.Value;
            tempDict.Add(name,value);
```

```
                // The last key of every node
                if (jP.Name.Equals(lastKey)) {
                    // If we do not copy the content of the
                        dictionary into another
                    // we will be copying by reference and
                        erase the content once
                    // we 'clear' the dict
                    Dictionary<string, object> copyDict =
                        new Dictionary<string, object>(
                        tempDict);
                    dictList.Add(copyDict);
                    tempDict.Clear();
                }
            }
        }
    }
```

Sin entrar mucho al detalle, lo que hace es parsear el objeto como objeto de tipo `JArray` y luego este se discretiza por cada elemento de esa cadena. Cada elemento representa a un nodo. Se recorre a su vez cada subelemento que lo conforma, que no son más que sus pares *llave-valor*.

Los datos que se han extraído se guardan en una lista de diccionarios de par `<string,object>`. Se usa la clase genérica `object` porque el tipo del valor asociado a la clave varía.

3. Se hace lo mismo para los enlaces. El proceso es similar.

4. A partir de los objectos extraídos de la deserialización del JSON, se **crean las instancias representantes** de los nodos del proyecto. Existe un problema de cierta gravedad en esto: en el JSON no existe ningún parámetro que explicite de que tipo de nodo estamos hablando. Es decir, que no podemos conocer directamente con qué aparato concreto estamos tratando.

   La solución que nosotros hemos tomado para sortear este problema es el añadir una etiqueta al nombre del nodo en el momento de la creación de la red en GNS3. Por ejemplo, si el nodo es un *VPC* (nodo predefinido y propio del simulador), su nombre sería de la forma "[VPC]NombreDelNodo". Será necesario entonces definir una etiqueta por cada tipo de aparato a utilizar.

   Sin embargo, sigue existiendo un problema de peso en todo esto: no conocemos el tipo de cada nodo antes del tiempo de ejecución (*runtime*, en inglés) del código. En otras palabras, hasta el momento de la descarga y el análisis del JSON es imposible saber con qué tipo de nodos estamos lidiando, con lo que es a su vez imposible escribir el constructor que se va a utilizar. Recordemos que nuestra intención es

usar una clase por cada modelo de nodo concreto. Por supuesto, se podría crear una sentencia condicional en la que, dependiendo de la etiqueta, llevara al constructor de una clase u otra. Sin embargo, requeriría de muchas líneas de código y sería necesario añadir más cada vez que se introduzca una nueva clase de modelo de aparato en la API.

La solución pasa entonces por lo que en programación se conoce como *reflexión*. Poniéndolo en términos puramente técnicos: "inspeccionar los metadatos y el código compilado en tiempo de ejecución se llama reflexión" [9].

Código 5.3: Instanciación de los nodos

```
System.Reflection.ConstructorInfo ctor; int i = 0;
try{
    foreach(Dictionary<string, object> node in JSON){
        try{
            Console.WriteLine($"Gathering information
                for node #{i}... ");

            // Get the main constructor of the node
                type
            ctor = Aux.NodeType(node["name"].ToString()
                ).GetConstructors(
                System.Reflection.BindingFlags.
                    NonPublic | System.Reflection.
                    BindingFlags.Instance
            ).Last();

            // Invoke the previous constructor and
                create the instance through it
            listOfNodes[i] = (Node)ctor.Invoke(
                new object[]{
                    node["console_host"].ToString(),
                    ushort.Parse(node["console"].
                        ToString()),
                    node["name"].ToString(),
                    node["node_id"].ToString(),
                    GetNodeListOfPorts(node)
                }
            );

            nodesByName.Add(listOfNodes[i].Name,
                listOfNodes[i]);
            nodesByID.Add(listOfNodes[i].ID,
                listOfNodes[i]);
        } catch(Exception err1){
            Console.Error.WriteLine(
                "Impossible to save the configuration
                    for the node #{0}: {1}",
                i.ToString(), err1.Message
            );
        }
```

```
            i++;
        }
    } catch(Exception err2){
        Console.Error.WriteLine(
            "Some problem occured while saving the nodes
                information: {0}",
            err2.Message
        );
        listOfNodes = null;
    }
```

Con esto en mente pues, tomamos el tipo de la clase asociada al aparato mediante la etiqueta que ya hemos mencionado. Obtenemos gracias a él el constructor que nos es útil. El siguiente paso es invocarlo para generar la instancia del objeto. Previamente se pensó en utilizar el método `Activator.CreateInstance()`, que instancia un objeto directamente con los metadatos del tipo representante (representados a su vez por la clase `System.Type`) y los parámetros del constructor, pero solo funciona cuando los constructores son públicos y no es el caso de aquellos que usamos.

Añadimos la instancia a una lista donde se guardan todos los objetos representantes de los nodos. Esta propiedad se verá más adelante.

La obtención de las interfaces que cada nodo posee así como de cuáles están libres y cuáles están usadas por cierto enlace requiere de más trabajo que no se considera necesario explicar aquí.

5. Es ahora el momento de instanciar los enlaces a través de la información extraída previamente. Aunque el problema que vimos con los nodos no se va a dar aquí, la obtención de los enlaces y sus parámetros tampoco es fácil. La principal dificultad aparece, una vez más, cuando se trata con las interfaces de los nodos.

Código 5.4: Matcheo entre los enlaces y las interfaces de los nodos

```
List<Dictionary<string,object>> dictList = null;
try{
    dictList = DeserializeJSONList(nodesJSON, "
        port_number");
} catch (Exception err){
    Console.Error.WriteLine(
        "Some problem occured while trying to gather
            information about the nodes connect to the
            link: {0}",
        err.Message
    );
}

if (dictList.Count > 0){
```

```
        // Iterates through the JSON dictionary
        foreach (Dictionary<string, object> nodeTemp in
            dictList){
            // Iterates through the nodes the link connects
            foreach (Node node in link.Nodes){
                if (node != null && node.ID.Equals(nodeTemp
                    ["node_id"].ToString())){
                    // Search for the port that matches the
                        found one
                    var foundPort = node.Ports.Where(
                        x => (
                            x["adapterNumber"].ToString()
                                == nodeTemp["adapter_number"
                                ].ToString() &&
                            x["portNumber"].ToString() ==
                                nodeTemp["port_number"].
                                ToString()
                        )
                    );
                    // If exists, add the link into the key
                        "link" of the ports list
                    // of dictionaries of the node
                    if (foundPort.Count() > 0) foundPort.
                        First()["link"] = link;
                }
            }
        }
    }
```

Una vez localizada la sección relacionada con los nodos en el diccionario que contiene la información de los enlaces extraída del JSON, se deserializa y se hace un barrido por entre esos nodos. Hecho esto, se busca entre los nodos que ya tenemos almacenados aquel que posea la misma ID que el del nodo sobre el que estamos barriendo. Si se encuentra, se busca la interfaz del mismo que dicte el nodo del barrido. De encontrarse, el puerto (usado aquí como sinónimo de interfaz) del nodo que teníamos almacenado guardará la información del enlace que contiene todos esos nodos que se están analizando. Es bastante confuso, sí.

El método principal que se encarga de la extracción de los enlaces, `GetLinks()`, se verá ayudado por un par de funciones declaradas localmente. Estas funciones locales aparecen por vez primera en C#7.

Se guardan todos estos enlaces como lista en otra de las propiedades que se verán más adelante.

6. Finalmente, se añade a cada nodo la información de los enlaces a los que está conectado.

Propiedades

- `ProjectID`, `Host` y `Port`: propiedades que se toman directamente de los parámetros que el constructor necesitaba.
- `NodesJSON` y `LinksJSON`: diccionarios que contienen los JSON descargados desde el servidor una vez han sido parseados. Por lo general esta propiedad no tendría porque ser usada por un usuario que no esté desarrollando la API, pero se mantiene como `public` por si hay algún caso en el que sí.
- `Nodes` y `Links`: posiblemente las propiedades más importantes de la clase. Son listas que contienen los objetos que representan los nodos y enlaces, respectivamente, del proyecto.

Métodos

La mayor parte de los métodos de esta clase son privados, pues son usados internamente como subrutinas de otros métodos más grandes. Sin embargo podemos encontrar algunos accesibles desde fuera de la clase:

- `StartNode()` y `StopNode()`: activan/desactivan un nodo del proyecto. Esto se consigue haciendo uso del método POST de REST hacia la URI correspondiente al nodo. En el código se lleva a cabo mediante la clase `System.Net.Http.HttpClient`, que provee las herramientas suficientes para enviar y recibir datos de un recurso web.

Código 5.5: Activación/desactivación de un nodo

```
// First part of the URL
string URLHeader = $"http://{host}:{port}/v2/projects/{
    projectID}/nodes";

// Pack the content we will send
ByteArrayContent byteContent = null;
try{
    string content = JsonConvert.SerializeObject(new
        Dictionary<string, string> { { "", "" } });
    byteContent = new ByteArrayContent(System.Text.
        Encoding.UTF8.GetBytes(content));
    byteContent.Headers.ContentType = new
        MediaTypeHeaderValue("application/json");
} catch(JsonSerializationException err){
    Console.Error.WriteLine("Impossible to serialize
        the JSON to send it to the API: {0}", err.
        Message);
}

if (byteContent != null){
```

```
        try{
            responseStatus = HTTPclient.PostAsync(
                $"{URLHeader}/{node.ID}/{status}",
                    byteContent
            ).Result.IsSuccessStatusCode;
        } catch(HttpRequestException err){
            Console.Error.WriteLine("Some problem occured
                with the HTTP connection: {0}", err.Message)
                ;
            responseStatus = false;
        } catch(Exception err){
            Console.Error.WriteLine("Impossible to {2} node
                 {0}: {1}", node.Name, err.Message, status);
            responseStatus = false;
        }
    } else{
        responseStatus = false;
    }
```

- `StartProject()` y `StopProject()`: activan/desactivan todos los nodos del proyecto. Se intentó paralelizar el activado/desactivado de los nodos sin resultado.

- `SetLink()`: pasándole los objetos representantes de dos nodos del proyecto, es capaz de descubrir cuáles de sus interfaces están vacías y, de haber, crea un enlace entre ellas. También se consigue mediante POST. Actualiza `Links` y otra serie de parámetros tras la inserción. Es un método de cierta longitud (algo más de 100 líneas sin contar otras definidas fuera de las que hace uso). En este método y en similares es muy recurrido el tipo `dynamic`, que obliga al compilador a averiguar el tipo real de la variable en el momento de ejecución del código. Algo similar a lo que ocurre en lenguajes interpretados como Python y Matlab.

- `EditLink()`: método polimórfico, pues dependiendo de si su parámetro es un `Link` o dos `Node` su comportamiento varía. Se parece a `SetLink()` pero este hace PUT y no POST a la URI. Ambas formas del método llaman a un método interno de `Link`. Tal y como su nombre indica, edita un enlace, permitiendo hacer variar los parámetros del mismo.

- `RemoveLink()`: con la misma base polimórfica que el anterior. Elimina un enlace del proyecto de GNS3 con un DELETE e inmediatamente a su representante objeto.

- `GetNodeByName()` y `GetNodeByID()`: dado un nombre o un identificador, respectivamente, devuelve el objecto representante de tal nodo.

5.1.2. Node

Constructor

El constructor principal de `Node` solo es llamado desde `GNS3sharp`. Es bastante sencillo: asigna parámetros básicos que la instancia de `GNS3sharp` toma del servidor. Entre ellos se encuentra la dirección del nodo. Gracias a ella y mediante otro método interno de la clase, se crea un cliente TCP y se establece un flujo de conexión para el envío y recepción de mensajes.

Código 5.6: Establecimiento de la conexión con el nodo

```
protected (TcpClient Connection, NetworkStream Stream)
    Connect(int timeout = 10000){
    // Network endpoint as an IP address and a port number
    IPEndPoint address = new IPEndPoint(IPAddress.Parse(this
        .consoleHost),this.port);
    // Set the socket for the connection
    TcpClient newConnection = new TcpClient();
    // Stream used to send and receive data
    NetworkStream newStream = null;
    try{
        newConnection.Connect(address);
        newStream = newConnection.GetStream();
        newStream.ReadTimeout = timeout; newStream.
            WriteTimeout = timeout;
    } catch(Exception err){
        Console.Error.WriteLine("Impossible to connect to
            the node {0}: {1}", this.name, err.Message);
        newConnection = null;
    }
    return (newConnection, newStream);
}
```

Especial atención a este método, que devuelve una tupla (recordemos, también nuevo en C#7) en lugar de una simple variable.

Para que solo clases que pertenecen a esta librería puedan hacer uso del constructor, se ha aplicado el modificador de clase `internal`, el cual solo permite llamadas desde el espacio de nombres donde esté definido. Encapsulamiento en estado puro. La API no permite la creación de nodos nuevos así que se ha optado por ocultar este método del desarrollador final.

No obstante, sí que se incluye un constructor-clonador. Se trata de un constructor de clase cuyo parámetro de entrada es otro `Node`, de forma que se replica enteramente en una nueva instancia.

Código 5.7: Clonador de nodos

```
public Node(Node clone){
```

```
        this.consoleHost = clone.ConsoleHost; this.port = clone.
            Port;
        this.name = clone.Name; this.id = clone.ID; this.ports =
             clone.Ports;
        this.tcpConnection = clone.TCPConnection; this.netStream
             = clone.NetStream;
    }
```

Propiedades

- `ConsoleHost` y `Port`: dirección y puerto donde el nodo está ubicado. Gracias a estos datos podremos establecer una conexión con el aparato.
- `Name` y `ID`: nombre e identificador único del nodo. El nombre debería comenzar por *[<EtiquetaDelNodo>]* para que `GNS3sharp` sea capaz de construir el objeto con la clase asociada a tal aparato.
- `Ports`: interfaces que posee el nodo. Es un diccionario de tres llaves: "adapterNumber", "portNumber" y "link". Los dos primeros son parámetros que GNS3 asigna a las interfaces. El último guarda la referencia del enlace asociado a la interfaz en caso de que esté siendo utilizada y `null` si no.
- `LinksAttached`: lista de `Link`. Referencias a los que el nodo está conectado.

Métodos

Esta clase destaca por sus dos métodos principales: `Send()` y `Receive()`.

- `Send()`: haciendo uso del flujo establecido durante la creación del objeto, se envía una cadena de texto previamente convertida a bytes. Antes de enviar cualquier mensaje, comprueba que es posible escribir en el canal.

Código 5.8: Envío de mensajes a un nodo

```
byte[] out_txt = Encoding.Default.GetBytes($"{message}\
    n");
this.netStream.Write(buffer: out_txt, offset: 0, size:
    out_txt.Length);
this.netStream.Flush();
```

- `Receive()`: algo más complejo que `Send()`, hace también uso del canal establecido, aunque necesita de pasos adicionales para gestionar correctamente la información que se recibe.

Cada mensaje recibido se convierte en una cadena de texto que va siendo añadida a una lista. A cada nueva recepción se consulta al canal si existe nueva información a leer; si no, se esperan unos segundos confiando en que existan datos a procesar por parte del servidor. Se vuelve a hacer la comprobación y, si efectivamente no queda nada más por leer, se analizan las líneas recibidas en busca de caracteres inválidos y se guardan.

Destructor

Esta es la única clase de la librería que hace uso de un destructor personalizado. Los destructores ayudan a definir las sentencias que serán ejecutadas junto antes de que el objeto en cuestión sea destruido (se desreferencie).

Únicamente se encarga de cerrar la conexión establecida con el nodo.

5.1.3. Herederos de Node

La clase `Node` no hace más que de padre para todo un abanico de clases. A partir de ahí se crea una estructura en árbol desde la que se ramifica una serie de subclases. Las tres herederas directas son `Router`, `Switch` y `Guest`, correspondientes a los tres amplios grupos de nodos que pueden encontrarse en GNS3. Estas tres clases son abstractas; su labor no es otra que la de declarar métodos importantes que cada uno de estos aparatos se espera que posean.

La definición de los métodos se encuentra en sus clases herederas, que no representan otra cosa que dispositivos concretos. Pongamos un ejemplo para que se vea más sencillo:

`Node` es la clase padre. Define el constructor, la forma de establecer la conexión con los nodos y los métodos para enviar y recibir mensajes de estos. De esta clase parte otra, `Router`, que por consiguiente es hija suya. Esta clase declara métodos abstractos que se espera que las clases representantes de los routers posean, como por ejemplo `GetIPByInterface()`, que dado el nombre de una interfaz ha de ser capaz de averiguar cuál es su IPv4 asociada. Finalmente, la clase `OpenWRT` va a heredar de `Router`. Definirá los métodos abstractos de la anterior y los suyos propios.

Es muy importante señalar que cada una de estas clases representantes de aparatos concretos deberán contar con una propiedad estática que contiene la etiqueta que el nodo deberá tener en su nombre. De nuevo con ejemplo: si queremos que el constructor de `GNS3sharp` pueda construir correctamente el objeto representante a un router OpenWRT de nuestro proyecto de GNS3, es necesario que a este se le ponga por nombre algo como

[OPENWRT]NombreDelRouter. De esta forma, automáticamente y una vez más haciendo uso de reflexión, la API es capaz de encontrar la clase a la que se hace referencia.

Código 5.9: Etiqueta de `OpenWRT`

```
private const string label = "OPENWRT";
/// <summary>
/// Label you must set in the name of the node at the GNS3
    project
/// <para>Name of the node must look like "[OPENWRT]Name"</
    para>
/// </summary>
/// <value>Label as a string</value>
public static string Label { get => label; }
```

Si se hace correctamente, se instanciará un objeto `OpenWRT`. De lo contrario, se optará por el genérico `Node`.

Este punto del trabajo es importante. Una de las cosas que más nos interesaban de la creación de la API era su reutilización por manos ajenas. La estructura establecida en ella permite que cualquiera que quiera hacer uso de ella y pretenda implementar un cierto tipo de nodos en su proyecto de GNS3 lo tenga fácil a la hora de crear un tipo que le dé soporte en la librería.

Actualmente GNS3 cuenta con implementaciones de docenas de dispositivos de gestión de red del mercado. Crear una clase representante de todas ellas es ciertamente un trabajo inviable. Sin embargo, se ha buscado que cada usuario que haga uso de ella lo tenga fácil para insertar la que necesite. Tan solo necesita añadir una clase que herede de su tipo de aparato correspondiente (`Router`...) y definir los métodos que le gustaría que esta tuviera.

El número de métodos definidos para cada clase va en función de las necesidades de cada uno. Un dispositivo real cuenta con cientos de funciones; abstraerlas todas como métodos es descabellado.

A continuación se muestra uno de los métodos que nosotros hemos generado para `OpenWRT`, clase que ha sido usada abundantemente en la integración final de este proyecto ya que el router al que hace referencia nos ha sido de gran utilidad.

Código 5.10: Método `GetIPByInterface()` de `OpenWRT`

```
public override string[] GetIPByInterface(string iface){

    string GetParameterIfconfig(string _iface, string type){

        string result = null; string command = null;
```

```
            if (type.Equals("IP"))
                command = $"ifconfig {_iface} | grep 'inet addr'
                    | cut -d: -f2 | awk '{{print $1}}'";
            else if (type.Equals("NETMASK"))
                command = $"ifconfig {_iface} | grep 'inet addr'
                    | cut -d: -f4 | awk '{{print $1}}'";
            if (command != null){
                string lineTemp;
                Send(command);
                foreach (string line in Receive()) {
                    lineTemp = line.Trim();
                    if (Aux.IsIP(lineTemp)){
                        result = lineTemp;
                        break;
                    }
                }
            }
            return result;

        }

        return new string[]{
            GetParameterIfconfig(iface, "IP"),
               GetParameterIfconfig(iface, "NETMASK")
        };

    }
```

El método se encarga de encontrar la IP relacionada con cierta interfaz del router. Básicamente, envía un comando al router mediante `Send()` y de la respuesta obtenida con `Receive()` extrae la IP.

La conclusión que ha de sacarse de esto es que todos los métodos, o la mayor parte de ellos, van a ser definidos en base a estas dos funciones, `Send()` y `Receive()`. Podemos llevar esta lógica algo más allá: si no existe método creado en cierta clase para realizar una operación que nos es necesaria, podemos hacer uso de aquellas dos funciones para, sobre la marcha, conseguir lo que esperamos.

5.1.4. Link

Constructor

Una vez más, el constructor es únicamente accesible desde el espacio de nombres y solo será llamado desde `GNS3sharp`. Se han definido dos concretamente: uno que supone que todos los parámetros del enlace de GNS3 son nulos (es un enlace ideal) y otro que supone que al menos uno de ellos es distinto.

Propiedades

- `ID`: identificador único del nodo. Como el resto de IDs, lo asigna automáticamente GNS3.
- `Nodes`: array de objetos `Node` que contienen los objetos representantes de los nodos que el enlace une en el proyecto.
- Parámetros del enlace: `FrequencyDrop`, `PacketLoss`, `Latency`, `Jitter` y `Corrupt`. Son parámetros que GNS3 permite para eliminar idealidades de las conexiones. Todos son números enteros.

Métodos

Esta clase solo cuenta con un método, `EditLink()`. Permite editar uno de los parámetros del enlace. Para ello, además de ser este alterado en el objeto, se modifica en el proyecto de GNS3 haciendo un PUT a la URI asociada al enlace. Ya se mencionó este método tratando uno de los de `GNS3sharp`.

5.1.5. Clases auxiliares

- `Aux`: define métodos de ayuda para otras clases del espacio de nombres como un identificador de direcciones IP. También se encarga de generar, mediante reflexión, un mapeo entre los tipos de nodos existentes en la API y la etiqueta asociada a ellos. Todo esto es realizado de forma automática.
- `RoutingTable`: define una estructura para las tablas de enrutamiento. Es usada por las clases herederas de `Router` para una gestión más eficiente de esas tablas. Dentro de la propia clase se define un `struct` usado para trabajar más cómodamente con cada una de las rutas de la tabla.
- `ServerProjects`: en esta clase deben definirse métodos asociados a la URI del server donde se lista el conjunto de proyectos definidos en él. Por el momento solo cuenta con un método (`GetProjectIDByName()`) que extrae el ID de un proyecto dado su nombre, realmente útil a la hora de instanciar `GNS3sharp`.

5.1.6. Estructura de la API

A continuación se muestra el diagrama UML de la API, desde el que puede observarse esquemáticamente el diseño de cada una de las clases y la relación que existe entre ellas.

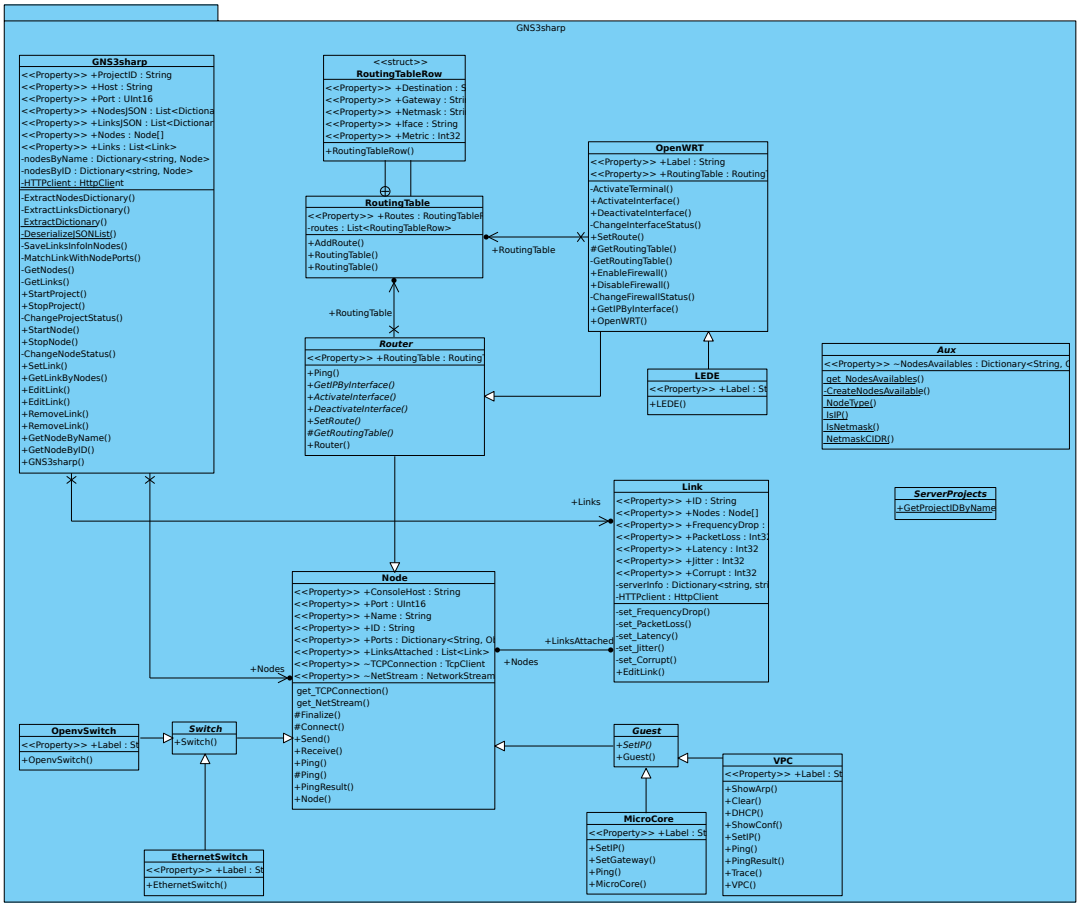


Figura 5.1: Diagrama UML de la API

5.1.7. Compilación

C# es un "lenguaje gestionado" (del inglés *managed code*) pues es compilado en código gestionado. Este código gestionado es representado en un lenguaje intermedio. Aparece entonces la figura del *Common Language Runtime* (CLR), núcleo del Microsoft .NET Framework. Se encarga de traducir este código intermedio al código nativo de la máquina desde la que se ejecuta.

Los contenedores de código gestionado se llaman ensamblados, y pueden ser archivos ejecutables (*.exe*) o bien librerías (*.dll*)[9]. Nuestro propósito es el de contener el código de la API en un ensamblado para librerías.

El editor de código Visual Studio de Microsoft tiene integradas decenas de herramientas para trabajar con proyectos dirigidos a .NET. Así, entre otras muchas cosas, facilita la compilación del código en C# para extraer de él un ensamblado que pueda ser incluido en otro proyecto como librería externa.

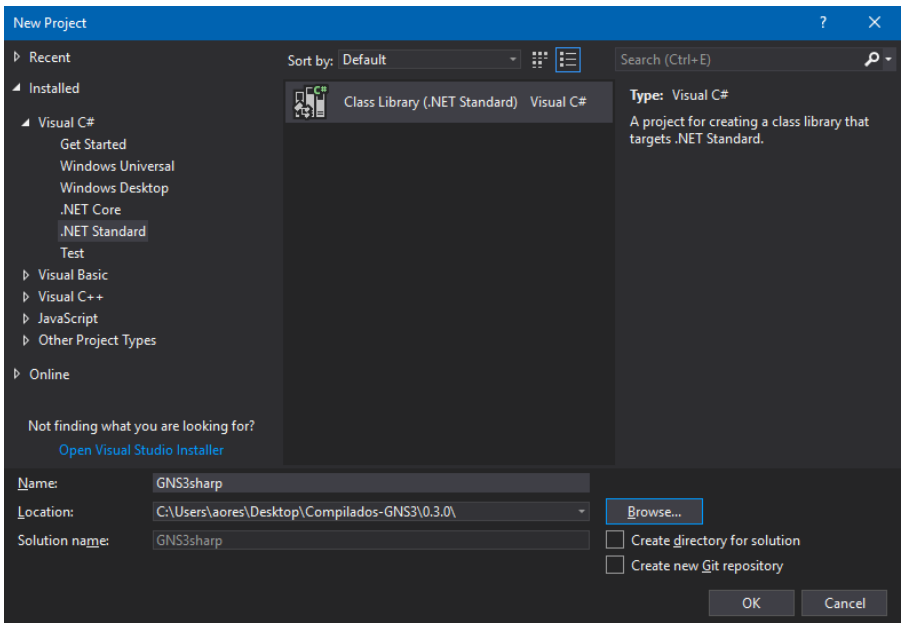


Figura 5.2: Elección del tipo de proyecto en Visual Studio 2017

Como framework con el que realizar la compilación se ha elegido *.NET Standard 2.0*. De esta forma la librería será portable y correrá sin modificaciones en versiones modernas de todos los frameworks principales de .NET (.NET Core, .NET Framework...) pues únicamente posee el núcleo común de todos ellos. En realidad, .NET Standard no es un Framework: es simplemente una especificación que describe una base mínima de funcionalidad (tipos y miembros), que garantiza la compatibilidad con un cierto conjunto de frameworks.

Para el correcto ensamblado del código de la API hará falta la importación de dos librerías externas: *Json.NET*, como ya se ha dicho en varias ocasiones, y *Microsoft.CSharp*. Esta última, no incluida por defecto en .NET Standard, añade funcionalidades para la compilación dinámica de código (el modificador de acceso `dynamic` o el propio instanciador de clases por reflexión).

Con el fin de facilitar a los desarrolladores la importación de código de terceros, Visual Studio integra una herramienta que permite instalar esas dependencias en tus proyectos a través de *NuGet*. NuGet es el formato de empaquetamiento de librerías de .NET para simplificar su compartición. Cuenta con una plataforma de hosting donde hospedar paquetes públicos paraser descrgados con facilidad por el resto de desarrolladores. La herramienta de Visual Studio se conecta directamente a este host, descarga el paquete y lo instala en el proyecto que esté siendo usado. Las dos librerías

señaladas anteriormente se encuentran en tal plataforma.

Establecidos una serie de parámetros que servirán de metadatos para el ensamblado, se compila el proyecto. Visual Studio crea entonces el archivo *.dll* así como un archivo *.xml* que, enlazado al ensamblado, aporta la documentación de nuestra librería. Para lograr esto último fue necesario la inclusión de etiquetas XML como encabezado de cada uno elemento de la API (clases, métodos, propiedades...) cuya sintaxis puede consultarse online. Similar al Javadoc de Java.

Ya tenemos preparada la librería para que Unity haga uso de ella.

5.1.8. GitHub y la comunidad

Hemos considerado esencial que la API sea reutilizable. El objetivo de su desarrollo en ningún caso ha sido el de nacer y morir para el presente trabajo. Lejos de esto, se pretende que esta librería pueda ser utilizada por quien quiera, ya tenga como propósito la construcción de un videojuego o cualquier otra aplicación.

Con esto en mente, todo el código utilizado por la librería se encuentra disponible en este repositorio de mi GitHub personal. Está liberado bajo una licencia MIT, tremendamente permisiva a la hora de reutilizar el código.

Además de los archivos de la propia API, se puede encontrar un README que explica algunos puntos importantes que considerar a la hora de utilizarla. En el repositorio de GitHub existe asimismo una sección de "releases", donde se suben los distintos compilados de la librería en formato ".dll" a medida que esta se expande o se reparan problemas aparecidos. Por supuesto, todo el código y su documentación están escritos en inglés para así llegar a más desarrolladores.

5.2. Desarrollo del videojuego

En esta sección se detalla la construcción del videojuego. En primer lugar se habla del juego en sí, qué se ha creado y qué no y qué se puede esperar de él. Inmediatamente después se describe el entorno generado en GNS3 para ser usado por Unity para, más adelante, mostrar cómo y qué ha sido desarrollado mediante él.

5.2.1. Descripción del juego

Aunque al comienzo el propósito del proyecto era desarrollar algo similar a lo descrito en la subsección 4.2.2, la complejidad existente para llevarlo

a cabo era demasiado grande. En su lugar, se ha preferido crear un único pequeño escenario que sirva como demostración del potencial del proyecto.

Con este propósito, se desarrollará un minijuego basado en tablas de enrutamiento. El objetivo del jugador es ser capaz de alcanzar un dispositivo marcado de la red en el menor tiempo posible. Por supuesto, habrá de atravesar una serie de nodos para alcanzarlo.

El mapa estará compuesto por varias plataformas y en cada una de ellas un cartel con la tabla de enrutamiento del respectivo router. Son estas plataformas entonces una suerte de representantes de los routers de la red. Al jugador se le darán varias posibles opciones como siguiente salto. Cada opción es una interfaz del router.

Con este nivel tan sencillo conseguimos dos cosas:

- Dar un ejemplo de juego didáctico con redes. Se ponen a prueba los conocimientos del jugador sobre redes y se le anima a hacerlo lo mejor posible a través de un contador ascendente.
- Mostrar cómo se lleva a cabo la interacción entre el simulador y el juego. Dicho en otras palabras, demostrar que, efectivamente, la API es funcional y puede ser usada con el propósito que se esperaba.

Se cimentan unos pilares sobre los que podrán construirse cosas mucho mayores en el futuro.

5.2.2. El proyecto de GNS3

Para empezar será necesario construir la red sobre la que se apoyará el videojuego. Se ha construido una pequeña, de apenas cinco routers y dos PCs, uno como plataforma de inicio y otro como destino a alcanzar.

La figura 5.3 está tomada directamente de la red desplegada en GNS3. Se comentarán a continuación los distintos aparatos usados y el modo en que han sido configurados.

Dispositivos usados

- **VPCs**: Los VPCs son unos de los pocos nodos interactivos que GNS3 incluye por defecto. Como su propio nombre indica, no son más que PCs virtuales que incluyen **funciones básicas** que se espera que un PC pueda usar en una red. Entre esas funciones está la de asignar una dirección a su interfaz y la de lanzar PINGs a través de ella. VPC es una gran herramienta para añadir hosts simples a GNS3 y probar la conectividad entre los nodos[11].

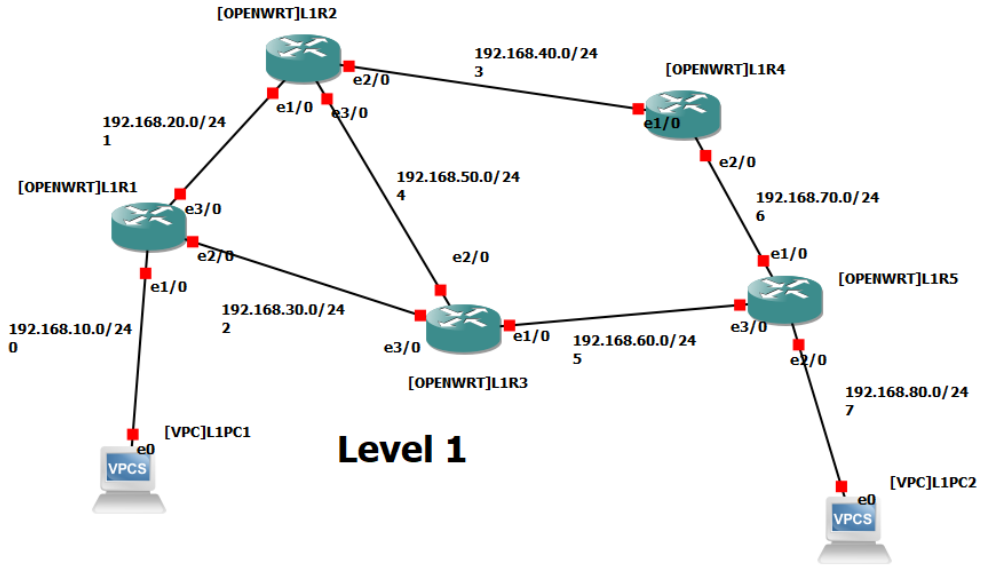


Figura 5.3: Red desplegada para el videojuego

Nuestra red cuenta con dos de ellos: *[VPC]L1PC1* y *[VPC]L1PC2*. Nótese la etiqueta con la que comienza su nombre, ya que como se comentó en anteriores ocasiones, esta sirve para que la API sea capaz de instanciar el objeto cuya clase representa este tipo de nodos (en este caso, la clase `VPC`). Solo consumen 2MB de RAM y la propia máquina anfitriona puede encargarse de gestionarlos (no es necesario pasar por GNS3 VM **ESTE CONCEPTO SERÁ EXPLICADO CUANDO HABLEMOS DE GNS3 AL COMIENZO**).

El único papel con el que cuentan estos dispositivos es el de representar mediante una dirección el inicio y el final del escenario.

- **OpenWRT**: OpenWrt es una distribución GNU/Linux altamente extensible para dispositivos embebidos (típicamente routers inalámbricos). A diferencia de muchas otras distribuciones para estos routers, OpenWrt está construido desde cero para ser un sistema operativo completo y fácilmente modificable para el router. Esto significa que es posible tener todas las características que son necesarias en estos dispositivos con el añadido que corresponde a contar con un núcleo Linux[12].

  La principal razón por la que se eligió esta arquitectura como dispositivo de enrutamiento para el proyecto es su **gratuidad**: el proyecto OpenWRT es open source liberado bajo una licencia GPL. Debido a esto y a su filosofía de "cada uno adapta su sistema operativo a sus

propias necesidades particulares", no se trata del firmware idóneo para un router al uso. Sin embargo, para nuestro propósito es más que suficiente.

Al estar basado en un sistema operativo Linux, gran parte de los comandos más famosos relacionados con redes como `ifconfig` aparecen aquí. Es un punto a su favor si conocemos tal arquitectura y las posibilidades que ofrece.

GNS3 cuenta con "plantillas" con las que se facilitar el importado de dispositivos como nodos a GNS3. Con el fin de incluir aparatos con OpenWRT instalado usamos la plantilla que puede descargarse desde aquí. Una vez descargada, siguiendo una serie de pasos GNS3 permite importar el nodo con mucha facilidad. La versión del firmware de OpenWRT que nosotros usamos es la 15.05.1. Slgo antigua considerando que data de 2016 y la última versión estable es la 18.06.1.

Todos los nodos que usen OpenWRT han de ser instalados en GNS3 VM, pues el simulador no permite de ningún modo que la máquina anfitriona sea la encargada de gestionarlos. La mayor parte de los aparatos cuyas imágenes son de cierta complejidad como este están obligados a ser usados de este modo.

Cinco de estos dispositivos han sido desplegados en la red. El camino más rápido para cruzar del primer PC1 al PC2 sería, en condiciones normales, *[OPENWRT]L1R1* → *[OPENWRT]L1R3* → *[OPENWRT]L1R5*.

Las redes que han sido marcadas en los enlaces de la figura 5.3 son únicamente orientativas ya que, como se explicará en el apartado siguiente, se ha pseudo-aleatorizado el establecimiento de estas. De nuevo, atención a *[OPENWRT]* en el nombre de los routers: es la etiqueta asociada a la clase `OpenWRT` para que `GNS3sharp` pueda instanciar el objeto apropiado.

- Otros dispositivos que consideramos añadir pero que, finalmente, por la simplicidad del diseño no se incluyeron, son el switch multicapa **Open vSwitch** y la variante de Tiny Core, distro Linux altamente modular, **Micro Core**. Ambos son dispositivos de uso libre y gratuito, de ahí nuestro interés por ellos.

Despliegue

Los dispositivos se han interconectado tal y como se aprecia en la ilustración 5.3. Ninguno de los enlaces cuenta con tipo alguno de filtro, con lo que se consideran ideales.

Por sencillez, todas las redes establecidas en el despliegue tiene máscara de red de 24 bits. Todas ellas son de la forma `192.168.x0.y`, donde `x`

es un número que se aleatoriza con cada arranque del juego mediante programación (hablaremos de esto cuando se detalle el apartado técnico de construcción del mismo) e `y` el número asociado a la interfaz del dispositivo que está conectada a esa red. Así, como ejemplo y suponiendo que las redes de la figura citada son las definitivas, a la interfaz de arriba de L1R1 (`eth3`) se le asignaría la dirección `192.168.20.3`. En el caso de los PCs, `y` sera en ambos casos `11`.

Al arrancar la red (al inicializar cada nodo), los VPCs son funcionales casi inmediatamente. No obstante, no es el caso en absoluto de los routers. Al tratarse de dispositivos que emulan aparatos reales, con un sistema operativo de cierta envergadura, es necesario **esperar para que su arranque sea completo**. Así, los nodos con OpenWRT necesitan una media de **HAZ AQUÍ PRUEBAS REALES** minutos en el PC desde el que se han realizado las pruebas para ser completamente funcionales. Esto implica que, o bien se ha de tomar la precaución de que todos los dispositivos estén iniciados al comenzar el juego, o bien que el jugador ha de esperar el tiempo necesario para que los aparatos comiencen a estar disponibles.

La problemática pasa a ser doble, ya que **GNS3 no guarda el estado de las máquinas tras su apagado**. Por más información que se ha buscado para intentar paliar este inconveniente, nada parece ser efectivo. La consecuencia natural es, por consiguiente, que el dispositivo habrá de ser configurado cada vez que se reinicie. Este no es un problema per se, pues gracias a la API puede llevarse a cabo mediante unas líneas de código con total facilidad; el verdadero problema es que, aún así, configurar todas las interfaces de un proyecto lleva tiempo. Se ha calculado cuánto tarda el de la nuestra y es **AÑADIR AQUÍ DATOS**. El tiempo podría ser minimizado si cada router es configurado paralelamente.

Todo lo anterior se resume en:

1. La asignación de direcciones se aleatoriza gracias a las posibilidades de scripting de la API.

2. Los routers requieren de un cierto tiempo de arranque, en absoluto inmediato, para ser completamente funcionales.

3. La configuración de los dispositivos es descartada cada vez que son apagados, con lo que es necesario reconfigurarlos tras cada arranque. Esto lleva implícito una inevitable inversión de tiempo.

5.2.3. El proyecto de Unity

En esta sección se describe tanto el diseño del nivel de nuestro juego como el proceso que se ha seguido para materializarlo. Se comenzará señalando

los materiales que han posibilitado la construcción del escenario.

Materiales

- El fondo es una imagen estática tomada de OpenGameArt, una famosa web que publica arte redistribuible para la elaboración de juegos. Está liberado bajo una licencia https://creativecommons.org/licenses/by/3.0/, lo que implica que puede ser compartido y copiado siempre y cuando se dé crédito al autor y una referencia al material. Un enlace desde donde puede ser encontrado ya se ha dejado en la línea anterior. Gracias a "Alucard" por esta imagen.

- El decorado del nivel se ha generado gracias a un conjunto de sprites localizados una vez más en OpenGameArt. En esta ocasión, el material tiene licencia CC0 1.0, de dominio público; totalmente libre.

  Un sprite es más un mapa de bits usado como elemento individual cuyo conjunto posibilita crear escenas en un videojuego. Si atendemos a esa definición y reparamos en la imagen a la que puede accederse desde el enlace anterior, es evidente que esa descripción no se corresponde a la figura. Digamos que esa imagen está formada por decenas de pequeños sprites y es tarea nuestra el extraerlos para nuestro propio uso. Unity nos lo pone fácil, ya que contiene una herramienta llamada "sprite editor" (visible en la figura 5.4), que disecciona imágenes para convertirlas en pequeñas piezas. Si los sprites están repartidos con un tamaño fijo en la imagen, como es nuestro caso, es tan sencillo como indicarle tal dimensiones en píxeles y Unity hará el resto.

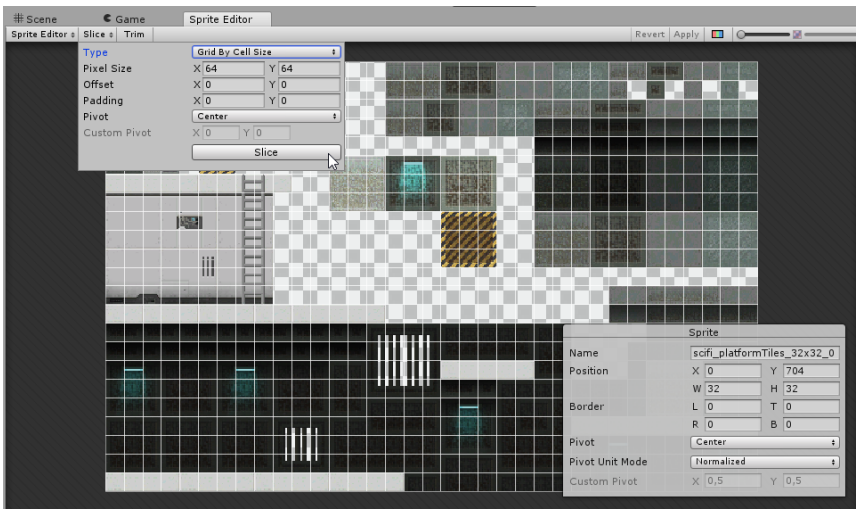


Figura 5.4: Editor de sprites de Unity

Ya veremos de qué modo son usados los sprites en el siguiente apartado.

- Finalmente, nos hemos visto ayudados por uno de los paquetes que Unity incluye por defecto. Se trata del paquete "2D", que puede ser descargado desde el propio cliente de Unity en *Assets* → *Import package* → *2D*. Incluye una serie de assets (**ESTE CONCEPTO DEBE EXPLICARSE LA PRIMERA VEZ QUE SE HABLE DE UNITY**) básicos para la creación de niveles bidimensionales.

Objetos del juego

El juego cuenta con, únicamente, dos escenas. La primera, aquella que aparece justo al ser inicializado, consta de únicamente una interfaz de usuario (GUI) que pide al jugador que pulse "espacio" para comenzar a jugar. Al hacerlo, se configura toda la red de GNS3 que pudimos ver en la figura 5.3. Todo el aspecto relacionado con la programación podrá estudiarse en el punto siguiente.

Pasado un tiempo, después de que todo quede configurado, Unity toma los datos necesarios del proyecto de GNS3 para construir la siguiente escena, la principal.

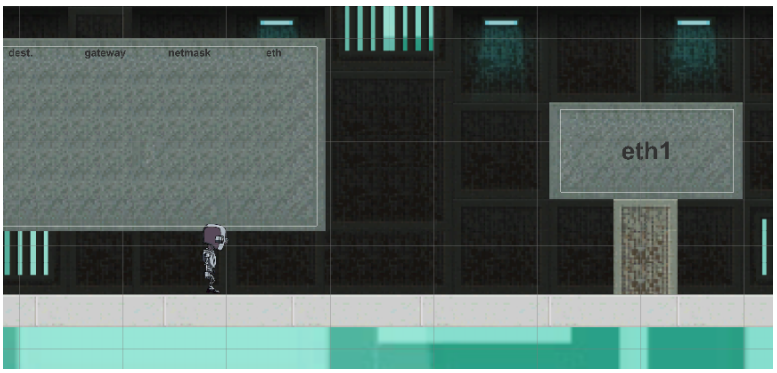


Figura 5.5: Plataforma del nivel del juego

En la figura 5.5 pueden apreciarse los elementos más importantes del escenario:

- Por un lado **el protagonista**, el objeto sobre el que el jugador tiene poder. Se controla como en cualquier juego de plataformas bidimensional (crucetas para moverse, espacio para saltar...). El personaje es uno de los assets que el paquete "2D" trae consigo, con lo que no fue necesario ningún tipo de programación adicional. Cuenta con físicas ya definidas a través de un elemento `Rigidbody 2D` (deja a un objeto

bajo el control del motor de físicas de Unity[13]) y un par de elementos collider (definen la forma de un objeto para las colisiones físicas[14]).

- Las **paredes** del escenario están construidas con sprites procesados como "tiles". Los tiles son objetos que permiten a un sprite ser renderizado en un "tilemap"[15]. Los tilemaps son, a su vez, una redecilla de cuadrículas donde se permite la inserción de estos tiles. Todo esto no es más que una herramienta que facilita la creación de niveles.

- El **suelo** del escenario también está formado por tiles. Sin embargo, la diferencia con los anteriores reside en que el tilemap sobre el que están montados contiene un elemento adicional: consta de un `Tilemap Collider 2D`. Este elemento otorga automáticamente físicas de colisión a todos los tiles que incluyamos en el tilemap. Ello significa que todo tile introducido en el tilemap desplegado colisionará con el protagonista, permitiendo a este moverse sobre él.

- Las **puertas** son la representación de las interfaces de los routers en el videojuego. Llevan a un nodo o a otro. Son, de nuevo, tiles, sobre los que se han dispuesto objetos vacíos que únicamente tienen un `Box Collider 2D` como elemento integrado. Posee un "trigger" (disparados) que manda una señal cada vez que un objeto colisiona con él.

- Los carteles de las **tablas de encaminamiento**. Necesitan ser llenados con la información que se extraiga directamente de los routers.

- Además, y aunque en la imagen no pueda ser apreciada, tenemos la cámara principal del juego, que rige lo que ve el jugador en todo momento. El objeto que la representa cuenta con un script predefinido en el paquete "2D" que permite seguir automáticamente a un objeto de la escena. Se ha elegido a protagonista como objeto que la cámara debe encargarse de seguir.

Para terminar este apartado, se facilita una vista más general del escenario en la figura 5.6, donde pueden observarse las distintas plataformas que lo conforman.

Scripting

El *scripting* de un videojuego no es más que la programación que existe en él para llevar a cabo toda la lógica que lo rige. Describiremos a continuación los distintos scripts que han sido utilizados para posibilitar el funcionamiento del juego:

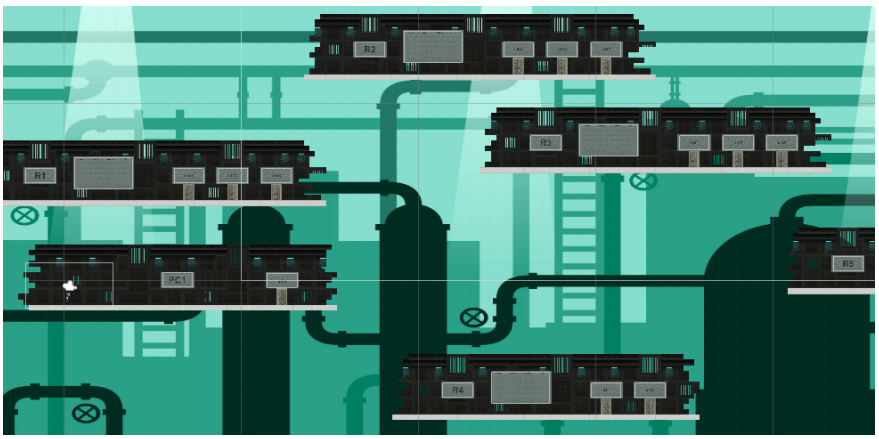


Figura 5.6: Escenario creado

- `GNS3Handler`: crea y guarda una instancia de `GNS3sharp` para ser utilizada por todos los objetos de la escena que la necesiten. Como no queremos que cada elemento de la escena instancie un objeto de esta clase (es un gasto de recursos innecesario), haremos que sea "singleton". En programación, un singleton es una clase que tan solo permite que **una instancia** de sí misma sea creada[16]. Para llevarlo a cabo, cada vez que se intente instanciar la clase, se comprobará si su propiedad estática `Instance` contiene información. En caso negativo, se crea el objeto; en caso contrario, se destruye el objeto de la escena desde el que se ha intentado instanciar la clase.

Código 5.11: `GNS3Handler`

```
using UnityEngine;

public class GNS3Handler : MonoBehaviour {

    public static GNS3Handler Instance { get; private
        set; } = null;
    public GNS3sharp.GNS3sharp projectHandler;

    // Awake is always called before any Start function
    void Awake() {
        if (Instance == null) {
            Instance = this;
            this.projectHandler = new GNS3sharp.
                GNS3sharp(
                GNS3sharp.ServerProjects.
                    GetProjectIDByName("GameNet")
            );
        }
        else if (Instance != this)  Destroy(gameObject)
            ;
```

```
        }

    }
```

Este script es colocado en la primera escena. Al arrancar esta, se instancia un objeto de la clase `GNS3sharp`, el cual solicita al servidor de GNS3 información acerca de un proyecto denominado "GameNet" (es como fue llamado el proyecto que contiene la red ya vista) para ser gestionado. Dada la simpleza del juego, no se ha creído conveniente habilitar mecanismos de control de errores, entendiendo por estos problemas a la hora de encontrar el proyecto solicitado, etc.

Bibliografía

[1] Wikipedia. Interfaz de programación de aplicaciones — Wikipedia, la enciclopedia libre. http://es.wikipedia.org/w/index.php?title=Interfaz%20de%20programaci%C3%B3n%20de%20aplicaciones&oldid=110157354, 2018. [Online; accedido el 24-August-2018].

[2] Google maps platform. Disponible en https://cloud.google.com/maps-platform/?hl=es.

[3] Pavol Návrat. Hierarchies of programming concepts: Abstraction, generality, and beyond. *Sigse Bulletink*, 26(3):17–28, 1994. Disponible en https://dl.acm.org/citation.cfm?id=187397.

[4] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. Disponible en https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[5] Esolang, the esoteric programming languages wiki. Disponible en https://esolangs.org/wiki/Hello_world_program_in_esoteric_languages.

[6] Joseph Hocking. *Unity in action*. Manning, 2000.

[7] Ignacio Roldán Etcheverry. Introducing c# in godot. Disponible en https://godotengine.org/article/introducing-csharp-godot.

[8] Armina Mkhitaryan. Why is c# among the most popular programming languages in the world? Disponible en https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb.

[9] Joseph Albahari and Ben Albahari. *C# 7.0 in a Nutshell*. O'Reilly Media, 7th edition, 2018.

[10] Microsoft. *Clase WebClient*, Octubre 2016. Disponible en https://msdn.microsoft.com/es-es/library/system.net.webclient(v=vs.110).aspx.

[11] Jason C. Neumann. *The Book of GNS3*. no starch press, 1st edition, 2015.

[12] About openwrt. Disponible en `https://wiki.openwrt.org/about/start`.

[13] Unity. *Rigidbody 2D*. Disponible en `https://docs.unity3d.com/Manual/class-Rigidbody2D.html`.

[14] Unity. *Colliders*. Disponible en `https://docs.unity3d.com/es/current/Manual/CollidersOverview.html`.

[15] Unity. *Tile*. Disponible en `https://docs.unity3d.com/Manual/Tilemap-ScriptableTiles-Tile.html`.

[16] Jon Skeet. *C# in Depth*. Manning, 3rd edition, 2013. Disponible en `http://csharpindepth.com/Articles/General/Singleton.aspx`.

[17] Microsoft. *What's new in C# 7.0*, Diciembre 2016. Disponible en `https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7#out-variables`.

[18] Microsoft. *An introduction to NuGet*, Enero 2018. Disponible en `https://docs.microsoft.com/es-es/nuget/what-is-nuget`.