



**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: emulador, GNS3, virtualización, videojuego, API

Resumen

En la época actual, la educación ha visto ampliadas sus fronteras hacia lo digital. Lo virtual ha propiciado un nuevo campo de entrenamiento para aprendices, que encuentran en ello una forma nueva de acceder al conocimiento. Sus posibilidades son muy amplias pues en el mundo digital cabe prácticamente cualquier cosa que de la imaginación salga.

Con este trabajo se pretende ahondar en la virtualización de redes para darle un nuevo propósito: acercarlo a la enseñanza. Además, este acercamiento pretende lograrse de un modo que si bien poco ortodoxo, cada día más sonado: los videojuegos. Así, el proyecto se propone fundamentalmente interconectar un emulador de redes con un motor de videojuego para aprovechar lo mejor de ambos elementos y llevarlo al ámbito educativo.

El documento recogerá el proceso llevado a cabo para lograr tal objetivo. Señalaremos las distintas tecnologías que nos ayudarían en la tarea, diseñaremos una solución con aquellas elegidas para ser a continuación implementada y estudiaremos el resultado obtenido.

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

Ángel Oreste, Rodríguez Romero

Keywords: emulator, GNS3, virtualization, videogame, API

Abstract

In the current era, education has seen its frontiers extended to the digital. Virtual world has provided a new training ground for apprentices, who find in it a new way of accessing knowledge. Its possibilities are very wide because in the digital world there is room for almost anything that comes from the imagination.

This work aims to deepen the virtualization of networks to give them a new purpose: to bring them closer to teaching. Moreover, this approach is intended to be achieved in a way that is unorthodox but increasingly popular: videogames. Thus, the project basically aims to interconnect a network emulator with a videogame engine to take advantage of the best of both elements and take it to the educational field.

The document will set out the process carried out to achieve this objective. We will point out the different technologies that would help us in the task, we will design a solution with those chosen to be implemented and we will study the result obtained.

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	v
1. Introducción	1
1.1. Introducción	1
1.1.1. Motivación	2
1.1.2. Objetivos	3
1.2. Estructura del trabajo	4
2. Planificación	5
2.1. Planificación temporal	5
2.2. Recursos empleados	5
2.2.1. Recursos humanos	5
2.2.2. Recursos software	6
2.2.3. Recursos hardware	7
2.3. Presupuesto	7
3. Estado del arte	9
3.1. Simuladores y emuladores de red	9
3.1.1. Mininet	10
3.1.2. Common Open Research Emulator (CORE)	10
3.1.3. GNS3	11
3.1.4. Más simuladores/emuladores	12
3.2. Motores de juego	12
3.2.1. Unreal Engine	12
3.2.2. GameMaker	13
3.2.3. Godot Engine	14
3.2.4. Unity	14
4. Análisis	15
4.1. Simulador de redes: GNS3	15
4.1.1. Arquitectura de GNS3	17
4.1.2. El servidor	17
4.1.3. La API	18

4.1.4. Nodos	18
4.1.5. Enlaces	19
4.2. Motor de videojuegos: Unity	19
5. Diseño de la solución	21
5.1. La API: GNS3sharp	21
5.1.1. Introducción	22
5.1.2. Elección del lenguaje para la librería	23
5.1.3. Diseño de la API	24
5.1.4. Diseño de clases	26
5.2. Diseño del videojuego	28
5.2.1. Interacción con el simulador	29
5.2.2. Modelo de videojuego	30
6. Implementación	33
6.1. Desarrollo de la API	33
6.1.1. GNS3sharp	33
6.1.2. Node	41
6.1.3. Herederos de Node	43
6.1.4. Link	45
6.1.5. Clases auxiliares	46
6.1.6. Estructura de la API	46
6.1.7. Compilación	47
6.1.8. GitHub y la comunidad	49
6.2. Desarrollo del videojuego	49
6.2.1. Descripción del juego	49
6.2.2. El proyecto de GNS3	50
6.2.3. El proyecto de Unity	54
7. Pruebas	63
7.1. Descripción de las pruebas	63
7.2. Caso 1	65
7.2.1. Consumo de RAM	65
7.2.2. Tiempo de arranque de los routers	65
7.2.3. Tiempo de establecimiento de la escena	65
7.3. Caso 2	66
7.3.1. Consumo de RAM	66
7.3.2. Tiempo de arranque de los routers	66
7.3.3. Tiempo de establecimiento de la escena	66
7.4. Conclusiones	66

8. Conclusiones	69
8.1. Trabajo realizado	69
8.2. Valoración del resultado	69
8.3. Experiencia personal	71
8.4. Trabajos futuros	71
Bibliografía	73

Índice de figuras

4.1. Interfaz de GNS3	16
4.2. Estructura interna de GNS3	17
5.1. La API en el proyecto	21
5.2. Boceto de diagrama UML de la API	26
5.3. Diagrama de la interacción entre Unity y GNS3	29
5.4. Boceto del hub del juego	31
6.1. Diagrama UML de la API	47
6.2. Elección del tipo de proyecto en Visual Studio 2017	48
6.3. Red desplegada para el videojuego	51
6.4. Editor de sprites de Unity	55
6.5. Plataforma del nivel del juego	55
6.6. Escenario creado	57
7.1. Comparación consumo de RAM	67
7.2. Tiempos de arranque de los nodos de forma individual (arriba) y en grupo (abajo)	67
7.3. Tiempos de arranque de las partes del juego relacionadas con el simulador	68

Capítulo 1

Introducción

Este primer capítulo introductorio sirve de acercamiento tanto al problema con el que se pretende resolver como al modo con el que se decidió resolverlo. Al final del mismo se listarán los distintos capítulos del documento así como un breve resumen de cada uno de ellos.

1.1. Introducción

El siguiente trabajo se propone desarrollar un **videojuego capaz de integrar un simulador de redes** en sus mecánicas; esto es, capaz de **verse afectado** por la configuración establecida en una red virtualizada y, a la misma vez, tener el poder de modificarla desde el propio juego. La finalidad de todo esto no reside en el entretenimiento puro que brinda el mundo del videojuego, sino en las puertas que abre para el mundo docente. Si consideramos a los videojuegos un medio para facilitar el aprendizaje, de hacerlo más atractivo al público, la integración de redes reales en ellos puede permitir crear un nuevo camino docente en el ámbito de las redes. Finalmente no es más que una forma de facilitar el aprendizaje para el mayor número de personas posible.

Se analizará una serie de tecnologías para seleccionar las más adecuadas para el trabajo. Como resultado de todo lo anterior, se proporcionará una biblioteca para que los desarrolladores puedan crear sus propios juegos sobre el simulador de red, y un juego de ejemplo de uso.

1.1.1. Motivación

Los videojuegos y la educación

Contrariamente al modelo educativo tradicional, en que el que la atención gira en torno al profesor, desde principios del nuevo milenio aparece el *aprendizaje centrado en el estudiante*. Aunque el origen del concepto puede datarse a principios del siglo pasado, no ha sido hasta este que ha comenzado a tomar forma. Carl Rogers, en su libro “Freedom to Learn for the 80s”, describe el cambio de poder del profesor experto al alumno, impulsado por la necesidad de un cambio en el entorno tradicional en el que, en esta “llamada atmósfera educativa, los estudiantes se vuelven pasivos, apáticos y aburridos” [1].

R.M Harden y Joy Crosby, en un artículo realizado en el 2000, describen las estrategias de aprendizaje centradas en el “maestro” desde el enfoque de *el maestro que transmite el conocimiento*, del experto al principiante. En contraste, describen el aprendizaje centrado en el estudiante como aquel enfocado en el aprendizaje de los estudiantes y “lo que los estudiantes hacen para lograrlo, en lugar de lo que hace el maestro”. Esta definición enfatiza el concepto del estudiante *haciendo* [2].

Es aquí donde los videojuegos pueden ayudar. Este mismo año, un informe salía a la luz dando una serie de datos bastante interesantes al respecto. El informe, desarrollado por Triseum (empresa dedicada de lleno a los videojuegos educativos), es un proyecto europeo con el propósito de probar y evaluar el aprendizaje basado en juegos con dos videojuegos educativos (ambos juegos creados por la propia empresa del estudio). De entre los 20 profesores que llevaron a sus aulas los juegos, ninguno de ellos tuvo experiencias negativas con ellos. Algunos de los más satisfechos con la prueba contaban que “los estudiantes estaban muy motivados y mucho mejor preparados para las siguientes asignaturas y conceptos de la clase, construidos mediante el contenido y los conceptos tratados en el juego” [3].

Cientos de estudios se han llevado a cabo con el fin de demostrar el potencial didáctico de los videojuegos. Algunos de ellos se han centrado en exponer el uso de juegos comerciales en escuelas [4], y otros tantos el de juegos nacidos por y para la educación [5]. Todos ellos, como el realizado por Kennedy-Clark y Thompson en 2011, concluyen que los beneficios de usar los videojuegos como método de aprendizaje funciona. Este último, asevera que “como se encontró en muchos otros estudios, el mundo virtual fue motivador, interesante y atractivo para los estudiantes” [6].

Y es que la virtualización es una herramienta enormemente versátil. Empresas como Labster se han dado cuenta de ello. Esta compañía se encarga de desarrollar laboratorios virtuales y propuestas lúdicas para la enseñanza

de materias relacionadas con la ciencia, muchas de ellas llevadas a cabo en forma de videojuego[7]. “Al integrar simulaciones en laboratorios virtuales permitimos que los alumnos tengan acceso ilimitado y económico y que practiquen todo lo que quieran hasta que aprendan los conceptos perfectamente”, dice su fundador, el danés Michael Bodekaer.

Enfoque del trabajo

Es entonces nuestra misión llevar esto último al terreno de las telecomunicaciones. A sabiendas de que los videojuegos suponen un portal educativo de gran riqueza, nos proponemos desarrollar un ejemplo de **juego que permita aprender telemática**.

La telemática “describe los procesos de transmisión y gestión de informaciones digitales así como los servicios y aplicaciones que se apoyan en ellos”[8]. Entre esos servicios se encuentran las redes. La infraestructura asociada a estas suele ser cara y costosa, requiriendo un desembolso importante así como un despliegue inicial no apto para cualquiera. Esto recuerda directamente a las palabras anteriormente expuestas por el fundador de Labster. Su solución fue virtualizar laboratorios e insertarlos en juegos para, así, acercar estos a los estudiantes. Nosotros aquí pretendemos algo similar.

El fin que se propone este trabajo es el de crear un videojuego que integre una red virtualizada. Por suerte para nosotros, a diferencia de Bodekaer, no tenemos que encargarnos de crear un sistema de virtualización de redes, pues, como veremos con detalle en el capítulo 3, ya existen varios. El objetivo principal pasa por enlazar tal simulador de redes (una herramienta de virtualización de las mismas) con un videojuego.

Precisamente aquel es el primordial problema con el que se habrá de lidiar. A día de hoy, no existe forma directa de unir un videojuego con un simulador de redes. Hay que considerar que estos simuladores están concebidos como finalidad en sí mismos (como una aplicación cualquiera) y no como herramienta para ser usada por utilidades externas. Se requiere por consiguiente desarrollar una infraestructura que haga de nexo entre la red virtual y un videojuego.

1.1.2. Objetivos

Aclarado lo anterior, los objetivos que el proyecto se propone son los siguientes:

- Posibilitar la **interacción directa entre un videojuego y un simulador de redes**. Aquí reside nuestro papel fundamental del proceso.

Con tal propósito crearemos una librería desde la que poder realizar la interacción.

- **Desarrollar un videojuego** que haga uso de tal librería; que sea capaz, asimismo, de encontrar un uso didáctico para ella y aplicarlo.

1.2. Estructura del trabajo

Para finalizar el capítulo, se expondrá a modo de adelanto una lista que recoge cada capítulo a tratar en el proyecto. Junto a todos ellos, se incluye un pequeño resumen que pretende dar a conocer qué podrá estudiarse en sus respectivas páginas:

- **Capítulo 1: Introducción.** En el actual capítulo, como ya se ha podido ver, se han abordado los distintas causas que han motivado a existir a este proyecto a la vez que la forma en la que se intentará llevar a buen puerto su resolución.
- **Capítulo 2: Planificación.** Una guía del trabajo llevado a cabo, concerniente tanto a la manera en que se ha distribuido temporalmente como a los utensilios empleados.
- **Capítulo 3: Estado del arte.** Las distintas tecnologías barajadas sobre las que trabajar son aquí listadas.
- **Capítulo 4: Análisis tecnológico.** Tras citar algunos ejemplos de tecnologías que podrían emplearse para el proyecto en el anterior capítulos, aquí se hace un análisis más en profundidad de las que han resultado escogidas.
- **Capítulo 5: Diseño.** Se determina la forma en la que se llevará a cabo la solución del problema inicial.
- **Capítulo 6: Implementación.** Trazado un camino a seguir en el capítulo anterior, en este se describe la manera en la que este camino toma forma. Se muestra qué se ha creado finalmente y cómo se ha logrado.
- **Capítulo 7: Pruebas.** En este capítulo el resultado final es mostrado y se añade una valoración del mismo.
- **Capítulo 8: Conclusiones.** Para finalizar el proyecto, se ha hecho un resumen del trabajo realizado junto a una valoración del mismo. Además, algunas propuestas de trabajos futuros sobre él son dadas.

Capítulo 2

Planificación

fgxdh

2.1. Planificación temporal

huigk

2.2. Recursos empleados

Dividiremos los recursos utilizados para la realización del trabajo en tres: recursos humanos, recursos software y recursos hardware:

2.2.1. Recursos humanos

Aquí una lista de las personas que han participado en el proyecto y su papel en el mismo.

- Juan José Ramos Muñoz, profesor del departamento de Teoría de la Señal, Telemática y Comunicaciones en la ETSIIT de la Universidad de Granada. Tutor del proyecto
- Jonathan Prados Garzón, profesor del departamento de Teoría de la Señal, Telemática y Comunicaciones en la ETSIIT de la Universidad de Granada. Cotutor del proyecto.
- Ángel Oreste Rodríguez Romero, alumno del grado de ingeniería de tecnologías de telecomunicación en la ETSIIT de la Universidad de Granada. Autor del proyecto.

2.2.2. Recursos software

Aplicaciones y demás material digital que ha sido necesario para desarrollar el trabajo:

- **Microsoft Windows 10 Home versión 1803** como sistema operativo sobre el que se ha llevado todo el proyecto a cabo.
- Se usará además **Xubuntu 18.04** instalado en una máquina virtual para llevar ciertas pruebas del capítulo 7 a cabo.
- **Mozilla Firefox Quantum** como navegador web. 61.0.2 es la última versión empleada.
- Para la realización de la memoria:
 - El editor **Texmaker 5.0.2** junto a MiKTeX 2.9, implementación de \LaTeX .
 - **LibreOffice Draw 6.1.0.3** para el diseño de algunos esquemas.
 - **InkScape 0.92** para la transformación de imágenes vectoriales en archivos de formato *.pdf*.
- Como lector de archivos *.pdf* ha sido elegido **Foxit Reader**. Última versión: 9.2.0.
- Para el desarrollo de código:
 - El editor más utilizado durante el transcurso del trabajo ha sido **Visual Studio Code**. Ha pasado por varias versiones durante su uso. La más reciente es 1.26.
 - Para la compilación de código y el scripting en Unity el hermano mayor del anterior, **Microsoft Visual Studio**, también ha pasado por varias versiones, pero la última de ellas ha sido la 15.8.0.
 - **Notepad++** ha sido utilizado para tomar notas y revisar código de forma rápida y ligera. La última versión usada es la 7.5.8.
 - Visual Paradigm Enterprise 15.1 para dibujar los diagramas UML.
- Para la virtualización de redes:
 - **GNS3 2.1.3 y finalmente 2.1.9** como simulador de redes.
 - **VirtualBox 5.2.16** para la virtualización de ciertos dispositivos de la red y la instalación de un sistema Linux (Xubuntu) en el que integrar GNS3.
- El motor para el desarrollo de videojuegos **Unity** versión personal. El último número de versión empleado ha sido el 2018.2.2f1.

- Para el capítulo 7 serán útil **RamMap 1.51** y **Process Explorer 16.21**.

2.2.3. Recursos hardware

Todo el proyecto ha sido construido a través del portátil Lenovo Z500. Cuenta con un procesador Intel i7 3232QM de ocho núcleos virtuales, 16GB de RAM DDR3 a 798MHz, un disco duro HDD de 1GB a 5400RPM y una tarjeta gráfica dedicada Nvidia 740M con 1GB VRAM.

2.3. Presupuesto

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. Este capítulo se encarga de hacer un listado de aquellas tecnologías que podrían ser útiles así como de dar una serie de apuntes acerca de cada una de ellas. Dividiremos las tecnologías en dos grandes secciones: simuladores/emuladores de red y motores de videojuego.

3.1. Simuladores y emuladores de red

Un simulador de red es un software que predice el comportamiento de una red de datos. En los simuladores, la red se modela con dispositivos, enlaces, aplicaciones, etc. y se informa del rendimiento de la red. La mayoría de los simuladores utilizan la simulación de eventos discretos (modelado de sistemas en los que las variables de estado cambian en momentos discretizados de tiempo). El comportamiento de la red y las diversas aplicaciones y servicios que soporta pueden ser observados en un laboratorio de pruebas; varios atributos del entorno también pueden ser modificados de manera controlada para evaluar cómo se comportarían la red bajo diferentes condiciones [9].

Sin embargo, conviene no confundir “simuladores” con “emuladores” de red. Un simulador puede realizar tareas replicadas para demostrar el comportamiento de una red y sus componentes, mientras que un emulador puede copiar el comportamiento de una red para reemplazarla funcionalmente [10].

En un nivel básico, un simulador de red utiliza fórmulas matemáticas para crear un modelo teórico y totalmente virtual de una red. Los simuladores son soluciones software. Si bien se utilizan principalmente con fines de investigación y educación, también pueden actuar como herramientas de prueba cruciales en el diseño y desarrollo de una red. Nosotros mostraremos ejemplos de ambos indistintamente. Solo citaremos aquellos gratuitos y/o

open source.

3.1.1. Mininet

Mininet es un emulador de red que crea una red de hosts virtuales, switches, controladores y enlaces. Utiliza la virtualización basada en procesos de forma que es capaz de ejecutar muchos nodos (según la web, ha logrado levantar incluso 4096 a la vez) en un único núcleo del sistema operativo. Desde su versión 2.2.26, Linux soporta **namespaces** de red, una característica de virtualización ligera que proporciona a los procesos individuales interfaces de red separadas, tablas de enrutamiento y tablas ARP (del inglés, *Address Resolution Protocol*). Mininet puede crear switches OpenFlow en el kernel o en el espacio de usuario, controladores para controlar los conmutadores y hosts para comunicarse a través de la red simulada. Se encarga de conectarlos usando pares de ethernet virtual (**veth**)[11].

Mininet presenta una serie de ventajas muy interesantes en el mundo de la virtualización de redes:

- Capacidad de alojamiento de miles de dispositivos en una misma red.
- Asegura un arranque rápido de los nodos.
- Tiene una instalación muy sencilla, pues es posible descargarse desde su web máquinas virtuales con todo preparado para que tan solo sea necesario instalarla en un hipervisor. También admite instalación nativa para Linux.
- Cuenta con una **API** (del inglés, *Application Programming Interface*) desde la que poder interactuar con la red y los dispositivos desplegados en ella. Sin embargo, está escrita en Python y no hemos dado con ningún motor de videojuegos que admita a este como lenguaje de scripting nativo.
- También posee una interfaz de usuario desde la que construir y gestionar la red.

Los últimos commits del [repositorio de Mininet](#) son anteriores al último mes, con lo que es claro que el proyecto sigue siendo mantenido.

3.1.2. Common Open Research Emulator (CORE)

CORE es una herramienta para construir redes virtuales. Como emulador, construye una representación de una red telemática real que funciona en tiempo real, a diferencia de las simulaciones de red, en las que como ya

vimos se utilizan modelos abstractos. La emulación en vivo puede conectarse a redes físicas y routers. Proporciona un entorno para ejecutar aplicaciones y protocolos reales, aprovechando las ventajas de la virtualización proporcionada por Linux o FreeBSD, únicos sistemas operativos donde está disponible.

Algunas de sus características clave son:

- Es eficiente y escalable.
- Ejecuta aplicaciones y protocolos reales sin modificaciones.
- Cuenta con una interfaz gráfica de usuario fácil de usar.
- Es altamente personalizable.

CORE se utiliza típicamente para la investigación de redes y protocolos, demostraciones, pruebas de aplicaciones y plataformas, evaluación de escenarios de redes, estudios de seguridad y aumento del tamaño de las redes físicas de prueba. Permite la inclusión de otras tecnologías de emulación de red como EMANE y ns-3 [12].

Sus características fundamentales son similares a las del emulador previamente citado. Al igual que aquel, CORE puede ser usado a través del GUI (la interfaz de usuario) o scripting mediante Python. Además, cuenta con una API propia. El demonio de Linux que CORE levanta escucha en un socket TCP para los mensajes de la API que lleguen de otra aplicación local o remota. La interfaz de usuario de CORE se conecta localmente a este demonio y utiliza la API para instanciar topologías. CORE también actuará como una “servidor de emulación” que escucha conexiones TCP de otro sistema. En el momento de la conexión, el otro sistema transmite mensajes al demonio de CORE, el cual puede controlar la emulación en tiempo real [13].

3.1.3. GNS3

GNS3 es un simulador gráfico de red multiplataforma open-source que se ejecuta en Windows, macOS y Linux [14]. Esto ya presenta una cierta ventaja respecto a sus anteriores competidores, cuya instalación solo es posible en Linux (y FreeBSD en el caso de CORE). Permite la virtualización de dispositivos reales tales como routers Cisco (para lo cual está especialmente preparado) y conectividad con otras redes reales como internet. Según su web, algunas de las marcas que utilizan este software son at&t, Intel, NASA o Verizon [15].

La aplicación crea un servidor en la máquina en la que es lanzado al cual se puede acceder mediante una API REST.

Cuenta con [documentación online](#), [foros muy activos](#) e incluso una [academia](#) donde se imparten cursos.

En el capítulo 4 hablaremos con más detenimiento de este simulador.

3.1.4. Más simuladores/emuladores

Existen más aplicaciones que virtualizan redes que no explicaremos aquí. Por citar algunas: Cloonix, Netkit, Shadow, VNX...

3.2. Motores de juego

Un motor de juego proporciona multitud de características que son útiles a través de muchos juegos diferentes, por lo que un juego implementado con ese motor obtiene todas esas características junto a elementos artísticos personalizados y código de juego específico para él [16].

Dividiremos cada uno de los que veremos en tres secciones fundamentales: plataformas hacia donde los juegos pueden ser exportados, características, lenguajes de programación que usan y precio.

3.2.1. Unreal Engine

Con una lista inmensa de juegos desarrollados con él [17], podemos asegurar que **Unreal Engine** (y más concretamente Unreal Engine 4, su última versión), de la compañía Epic Games, es uno de los motores de juego más usados de la historia. Es por eso, y por las características que veremos a continuación, que aparece en las listas de motores más famosos.

- **Plataformas soportadas:** Windows PC, PlayStation 4, Xbox One, macOS, iOS, Android, AR, VR, Linux, SteamOS y HTML5. El editor tiene soporte en Windows, Linux y macOS [18].
- **Características:** renderización fotoreal en tiempo real, efectos de calidad de película de post-procesamiento, un flexible editor de materiales, abundantes herramientas de animación...[19] Unreal no ha llegado a ser un motor tan importante en vano. Quizá, de entre todas sus prestaciones, la más interesante sea los “blueprints”. Se trata de una forma visual y amigable de programar. Es posible crear rápidamente prototipos y contenido interactivo sin tocar una línea de código. Puede ser utilizado para construir comportamientos e interacciones de objetos, modificar interfaces de usuario, ajustar controles de entrada y mucho más.

- **Lenguajes de scripting:** aunque anteriormente era posible usar UnrealScript como lenguaje para desarrollar en Unreal, a partir de la versión 4 se decidió eliminar en favor de **C++**, que pasaría a ser el único lenguaje compatible con el software. Cuándo le preguntaron acerca de este cambio, Tim Sweeney, fundador de Epic Games, respondió: “¿Por qué no matamos a UnrealScript y pasamos a C++ puro? Ya sabes, máximo rendimiento y máxima depuración. Nos da todas estas ventajas” [20].
- **Precio:** 5 % de royalties sobre los ingresos brutos después de los primeros \$3000 por producto y trimestre fiscal [18].

Pese a los blueprints, que facilitan enormemente el trabajo lógico del juego, la curva de aprendizaje de Unreal Engine es algo mayor que para otros motores de juego, tal y como coinciden en múltiples foros de internet [21]. El hecho de trabajar con C++, un lenguaje antiguo y de bajo nivel, ya supone un incremento de dificultad.

3.2.2. GameMaker

Otro de los motores que suele aparecer en listas entre los más populares es **GameMaker Studio 2** (en su última versión) [22]. Su ventaja fundamental sobre la mayor parte de los motores es que no requiere de conocimientos de programación, lo que lo hace ideal para gente más comprometida con el puro diseño. Juegos condecorados por la crítica como Hyper Light Drifter y Undertale han sido desarrollados a través de este motor [23].

- **Plataformas soportadas:** a partir de su versión 2, es posible ser usado tanto desde Windows como de Mac. Soporta exportación a Windows desktop, Mac OS X, Ubuntu, Android, iOS, fireTV, Android TV, Microsoft UWP, HTML5, PlayStation 4 y Xbox One, aunque muchas de las plataforma de exportación solo puede ser accedidas pagando [24].
- **Características:** múltiples herramientas para trabaja con tiles y tilesets... Sus características no parecen destacar frente a las del resto de lenguajes [25]. Está orientado mayormente a juegos en dos dimensiones.
- **Lenguajes de scripting:** aquí es donde GameMaker brilla. Y es que si no sabes cómo programar, no hay ningún problema, ya que GameMaker cuenta con un sistema llamado “Drag and Drop” (arrastra y suelta) que permite al desarrollador crear un juego sin escribir una

sola línea de código [25]. Si se quiere tomar el control del juego mediante programación también es posible mediante el lenguaje propio del motor llamado GML [26], basado en C.

- **Precio:** aunque el motor puede usarse y probarse de forma gratuita, es necesario pagar para publicar los juegos. En función de la plataforma elegida el precio oscila [24].

Aunque por un lado su “Drag and Drop” es una característica extremadamente interesante, para nuestro propósito es más bien inútil. Incluso si quisiéramos usar su lenguaje propio, perdemos en flexibilidad (cualquier cosa que desarrollemos solo podrá ser usada para este motor) y, además, nos dificulta el trabajo, pues no se trata de un lenguaje plenamente desarrollado y, por ende, no cuenta con las herramientas que otros tantos sí.

3.2.3. Godot Engine

De entre los motores de juego propietario, aparece **Godot** con su propuesta open-source. La última versión, Godot Engine 3, apareció a finales de enero con un nuevo motor de renderizado, flujo de renderizado basado en físicas para 3D, soporte para C#... y otra serie de novedades [27].

- **Plataformas soportadas:** Windows, macOS, Linux, FreeBSD, OpenBSD y Haiku. Es posible exportar los juegos a iOS, Android, Windows, macOS, Linux, UWP, *BSD, Haiku e incluso a HTML [28].
- **Características:** creación de tus propias herramientas, una arquitectura que combina lo mejor de la renderización directa con la eficiencia de la renderización diferida, iluminación global para gráficos en tiempo real, edito de “tile-maps”, transiciones personalizadas para crear animaciones...[28]
- **Lenguajes de scripting:** Godot tiene un lenguaje propio, muy similar a Python, llamado GDScript. Su parecido con Python equivale a sencillez de desarrollo. Además, soporta C++ y, desde su última versión, C# [28]. Se está trabajando asimismo en la [implantación de Python](#) como lenguaje de scripting nativo.
- **Precio:** quizás el punto fuerte de Godot: es totalmente gratuito, open-source y además está liberado bajo una licencia MIT, sin restricciones [29].

Pese a no estar soportado de la forma en que un motor propietario de Unreal puede estarlo, Godot es un motor realmente interesante con prestaciones más que suficientes para el trabajo que nosotros pretendemos llevar a cabo.

3.2.4. Unity

Capítulo 4

Análisis

En el capítulo anterior se listaron distintas tecnologías que podían sernos útiles para nuestro proyecto. En el actual nos detendremos a explicar cuáles de ellas explicaremos y el por qué de su elección.

4.1. Simulador de redes: GNS3

Se ha decidido que GNS3 sea el simulador de redes a utilizar para nuestro proyecto por varias razones:

- Es **multiplataforma**, con lo que podemos trabajar sencillamente con él tanto desde Windows como Linux.
- Existe **mucha información** para consultar. Desde internet contamos con documentación oficial y un foro propio. Hay incluso libros sobre él y como ejemplo tenemos el usado en varias ocasiones en este documento, “The Book of GNS3”, por Jason C. Neumann.
- Es altamente **expandible**: hay decenas de aparatos reales que pueden incluirse y emularse en las topologías creadas. Pueden descargarse desde [su marketplace](#).
- **Cuenta con una API REST** que lo hace interactivo desde el exterior.
- Integra **WireShark** nativamente para estudiar el envío de datos entre los distintos dispositivos.

Con la descripción de simulador y emulador de red que pudo verse en la sección 3.1, entendemos que GNS3 es un simulador (tal y como es descrito en la mayoría de fuentes) que incluye en sí mismo una serie de emuladores que

ayudan a recrear los entornos. Las emulaciones son posibles gracias a herramientas como QEMU (un emulador de máquinas y virtualizador open source [30]). Lo que ocurre entre los distintos dispositivos (emulados) desplegados en GNS3 es lo que el software simula.

GNS3 se estructura en **proyectos**. En cada proyecto podemos desplegar una serie de dispositivos y conectarlos entre sí. Al guardar un proyecto como tal, el despliegue realizado se guarda junto a él. Sin embargo, el estado de las máquinas no corre la misma suerte, y es que a cada apagado de las mismas toda su configuración se pierde.

Una de las características más interesantes de GNS3 y sus proyectos está en la importación y exportación de estos. El simulador cuenta con una opción que permite extraer todo un proyecto en una imagen (de formato *.gns3project*) para que sea fácilmente portable. Existe la posibilidad de exportar, junto a la topología creada, las imágenes base sobre las que los nodos del proyecto están construidas para que no sea necesario contar con esas imágenes en el GNS3 donde se pretende importar.

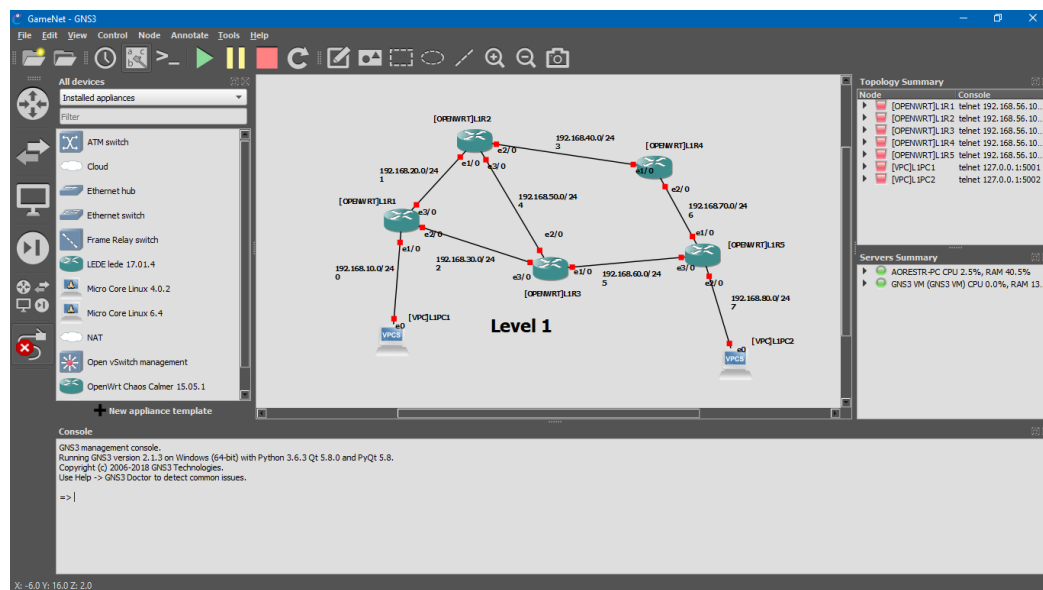


Figura 4.1: Interfaz de GNS3

La interfaz de usuario de la aplicación puede verse en la figura 4.1. La barra de arriba está repleta de opciones relacionadas con el proyecto, como arrancar todos los dispositivos, pausarlos, pararlos o incluso herramientas de dibujo para convertir el esquema en algo más intuitivo y legible. A la izquierda se listan los nodos disponibles en la máquina. Arrastrándolos al centro los insertamos en el proyecto.

A continuación pasaremos a explicar conceptos clave del emulador.

4.1.1. Arquitectura de GNS3

La documentación de GNS3 recoge el modo en el que la aplicación está estructurada. La ilustración 4.2 está sacada directamente de ella [31].

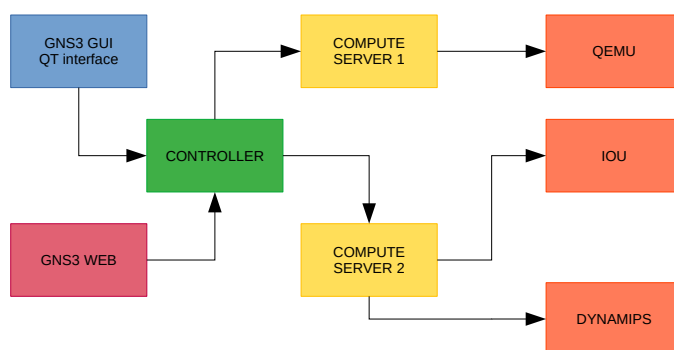


Figura 4.2: Estructura interna de GNS3

Las distintas partes que componen la aplicación se transmiten la información sobre HTTP haciendo uso de mensajes JSON (del inglés, *JavaScript Object Notation*), un formato de intercambio de datos ligero fácil de leer y escribir para humanos [32].

Volviendo a la imagen, el controlador (*controller*) controla toda la parte que gestiona el estado de un proyecto y lo guarda en el disco. Sólo existe un controlador para toda la aplicación. El GUI o interfaz de usuario es la encargada de mostrar las topologías. Toda la información que necesita la toma directamente del controlador. En los “compute” es donde se ejecutan los emuladores. Para cada aparato de la topología se inicia una instancia de emulador.

4.1.2. El servidor

El controlador y los “compute” de GNS3 forman el servidor del mismo. GNS3 aprovecha la tecnología cliente-servidor; al igual que un navegador web se conecta a un servidor web para acceder a las páginas web y mostrarlas, el GUI accede al servidor, permitiendo que se inicie, pare y controlar los dispositivos desplegados. Esto permite que los proyectos sean fácilmente escalables, ya que no necesitan ser ejecutados en un único equipo. Si se pretende trabajar con topologías grandes o complejas, también se puede ejecutar el servidor GNS3 en un PC diferente a aquel donde el GUI es ejecutado. Por consiguiente, de contar con un servidor de altas prestaciones, es posible instalar el servidor de GNS3 allí y desde otro sistema remoto construir y controlar las redes [14].

4.1.3. La API

GNS3 habilita la interacción con el servidor mediante una API REST (cuyo significado será explicado en la sección 5.1.1). Mediante el protocolo de red HTTP es posible construir proyectos y llenarlos con nodos y enlaces (que se definirán a continuación) sin necesidad de hacer uso de la interfaz de usuario. En la breve [documentación de la API](#) oficial es posible ver algunos ejemplos de uso.

GNS3 se encarga de asignar a cada proyecto un ID único. Un ejemplo de recurso puede ser el siguiente, donde se accede a un JSON con los proyectos abiertos en GNS3 e información básica sobre ellos: *http://<IP del server>:<puerto del servidor>/v2/projects*.

4.1.4. Nodos

Cada dispositivo 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. Por norma general, las virtualizaciones se realizan a partir de 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.

Tanto en macOS como en Windows es necesario utilizar **GNS3 VM** para hacer uso de ciertos nodos. Se trata de una imagen con un Ubuntu ligero que se instala en un hipervisor (VMware, Virtualbox) y hace de sistema operativo desde el que ejecutar los aparatos que se implanten en la red. Las ventajas de usar GNS3 VM son las siguientes [33]:

- Algunas dependencias son difíciles de instalar, como los requisitos para IOU; necesitan una librería específica.
- Con VMware es posible utilizar utilizar la aceleración KVM para QEMU.
- Dynamips y QEMU funcionan mejor en Linux (menos problemas aleatorios con ASA por ejemplo).
- Soporte completo de IOU.
- No hay antivirus o firewall dentro de la máquina virtual que modifique el tráfico de la red.
- La máquina está aislada del host, lo que implica un mayor nivel de seguridad.

Los nodos pueden ser inicializados juntos (todos los del proyecto a la vez) o por separado. GNS3 los asigna a un puerto de la red desde el que pueden ser accedidos y controlados mediante telnet.

En la API, para obtener los distintos nodos que existen un proyecto es necesario acceder al recurso: `http://<IP del server>:<puerto del servidor>/v2/projects/<ID del proyecto>/nodes`

4.1.5. Enlaces

Los nodos están interconectados entre sí mediante enlaces. Los enlaces son los encargados del envío de paquetes entre ellos. Sus características pueden ser modificados mediante “filtros de paquete” (*Packet filters*), que no son más que unos parámetros que afectan a las conexiones. Desde la latencia a la probabilidad de pérdida de paquetes se puede controlar mediante ellos.

El recurso asociado en la API a los enlaces de un proyecto es: `http://<IP del server>:<puerto del servidor>/v2/projects/<ID del proyecto>/links`

4.2. Motor de videojuegos: Unity

Una de las principales razones por las que se ha elegido Unity como el motor sobre el que desarrollar nuestro juego es por su popularidad y su facilidad de aprendizaje. Cuenta con toneladas de información en la red y con foros muy activos que permiten encontrar respuesta a las preguntas generadas a gran velocidad. Pero Unity es una buena elección también por méritos propios.

El motor posee simulación física propia, oclusión ambiental (SSAO), sombras dinámicas... y una larga lista de elementos más. Muchos motores de juego cuentan también de estas características, pero Unity tiene dos ventajas principales sobre otras herramientas de desarrollo de juegos igualmente avanzadas: un flujo de trabajo visual (*visual workflow*) extremadamente productivo y un alto grado de compatibilidad entre plataformas.

El workflow visual posee un diseño bastante único, diferente de la mayoría de los otros entornos. El editor se utiliza para diseñar las escenas de su juego y para unir los “assets” (cuyo significado veremos más adelante) de arte y el código en objetos interactivos. Lo bueno de este editor es que permite crear juegos de calidad profesional de forma rápida y eficaz, proporcionando a los desarrolladores herramientas increíblemente productivas a la vez que utilizando las últimas tecnologías en el mundo de los videojuegos.

Capítulo 5

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.

5.1. 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.

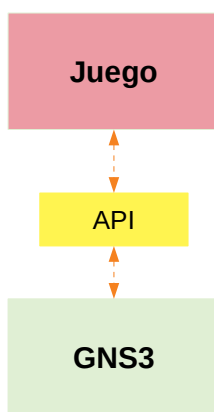


Figura 5.1: La API en el proyecto

Esta librería será usada mediante una API, que en nuestro proyecto se encuentra, metafóricamente, entre el juego y GNS3 tal y como se deja entrever en la figura 5.1. Todos sus detalles serán explicados en las próximas

páginas.

5.1.1. 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”[34]. 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[35]. 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[36]. 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 existen 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[37].

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.

5.1.2. Elección del lenguaje para la librería

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[38]. 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[16]. En general, C# es el lenguaje usado por defecto en Unity y el recomendado por todos, documentación incluida.
- **Porque es utilizado por varios motores.** 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[39]. 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, se trata de un lenguaje versátil y robusto.** 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[40]. 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 estático**, 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[41].

Particularmente, en este trabajo adoptaremos 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.

5.1.3. 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 los distintos aspectos que la compondrán (estructura, servicios, funciones...). 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 5.2.1 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`[42], 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!

5.1.4. 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:

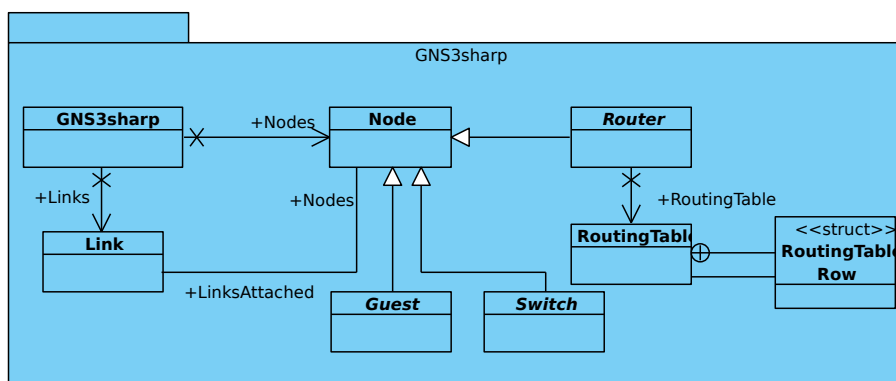


Figura 5.2: Boceto de diagrama UML de la API

El diagrama UML de la figura 5.2 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 **Node** y que a su vez esta es usada por **GNS3sharp** por alguna propiedad llamada **Nodes** (que hasta la subsección 6.1.1 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 5.1.3, 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 esa 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 5.2, 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 5.2).

5.2. 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.

5.2.1. Interacción con el simulador

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.

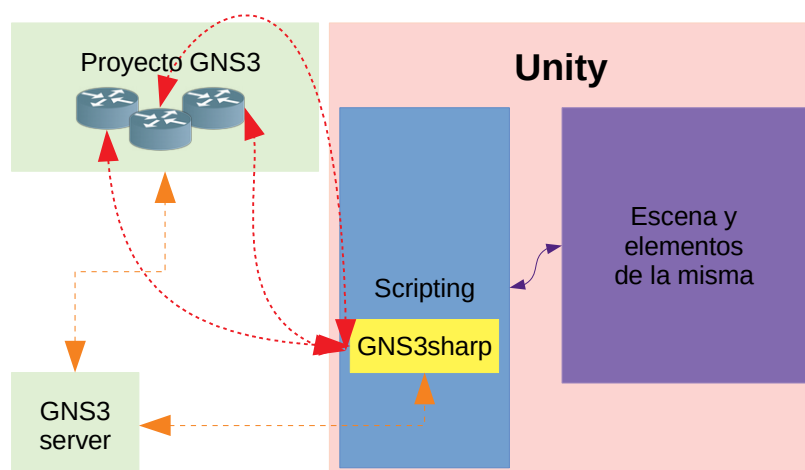


Figura 5.3: Diagrama de la interacción entre Unity y GNS3

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 declararan 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 5.3. El bloque morado hace referencia a todo el entorno que comprende una escena de Unity. Esta escena contendrá una serie de `GameObjects`, 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 `GameObjects` de la escena no tendrán más que usar esa instancia para poder conectarse a los nodos (entre otras cosas) de la simulación.

5.2.2. 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.

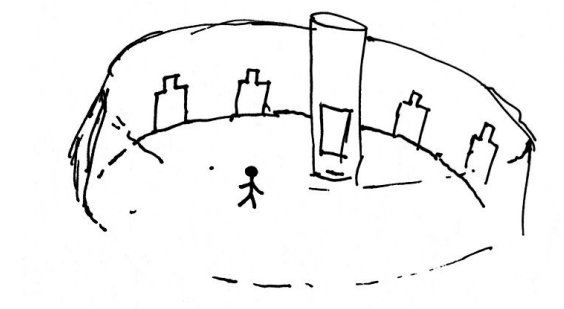


Figura 5.4: Boceto del hub del juego

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 inmediato. 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 dificultades encontradas en el camino que han frenado de una forma u otra el escenario inicial.

Capítulo 6

Implementación

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.

6.1. Desarrollo de la API

Si bien en la sección 5.1 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.

6.1.1. 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 6.1: 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 6.1.5, 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 6.2: Deserialización de JSON

```
// JSON array object
JSONArray jsonArray = JSONArray.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 (JsonObject jO in jsonArray.Children<JsonObject>
        >()) {
        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 objetos 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 explicita 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” [41].

Código 6.3: 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 occurred 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 6.4: 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 occurred 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 6.5: 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 occurred
        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 objeto representante de tal nodo.

6.1.2. 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 6.6: 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 6.7: 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 6.8: 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.

6.1.3. 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 6.9: 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 6.10: 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.

6.1.4. 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**.

6.1.5. 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**.

6.1.6. 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.

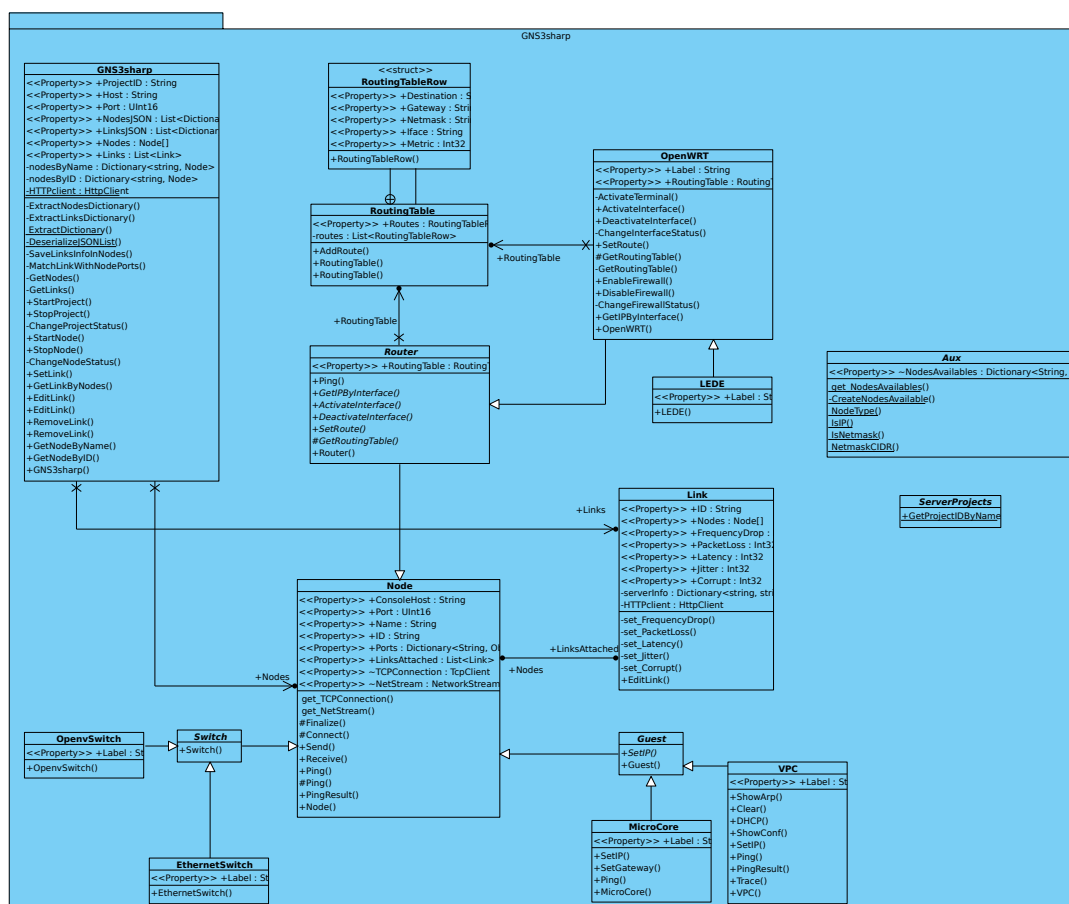


Figura 6.1: Diagrama UML de la API

6.1.7. 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)[41]. 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.

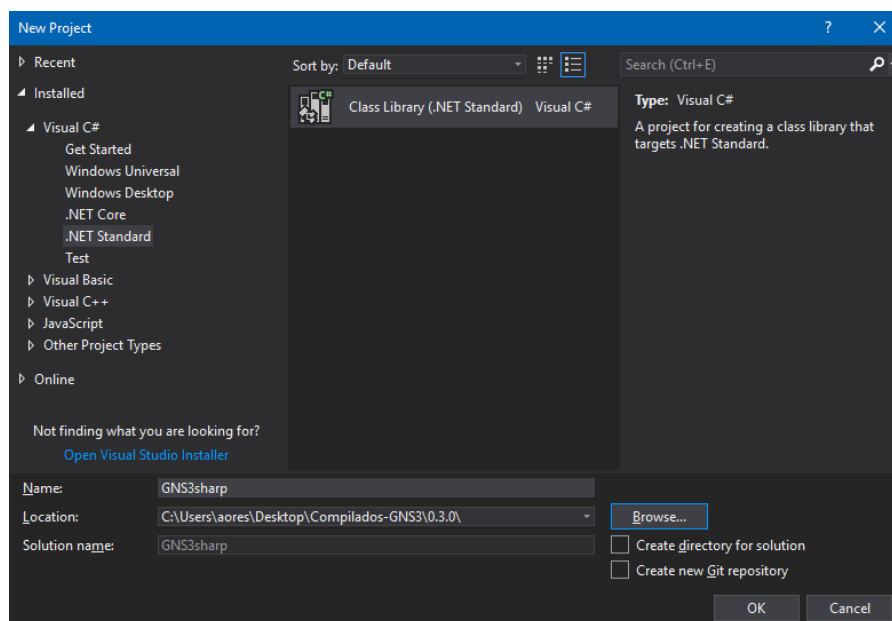


Figura 6.2: Elección del tipo de proyecto en Visual Studio 2017

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 para ser descargados 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.

6.1.8. 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.

6.2. 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.

6.2.1. Descripción del juego

Aunque al comienzo el propósito del proyecto era desarrollar algo similar a lo descrito en la subsección 5.2.2, 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.

6.2.2. 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 6.3 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[14].

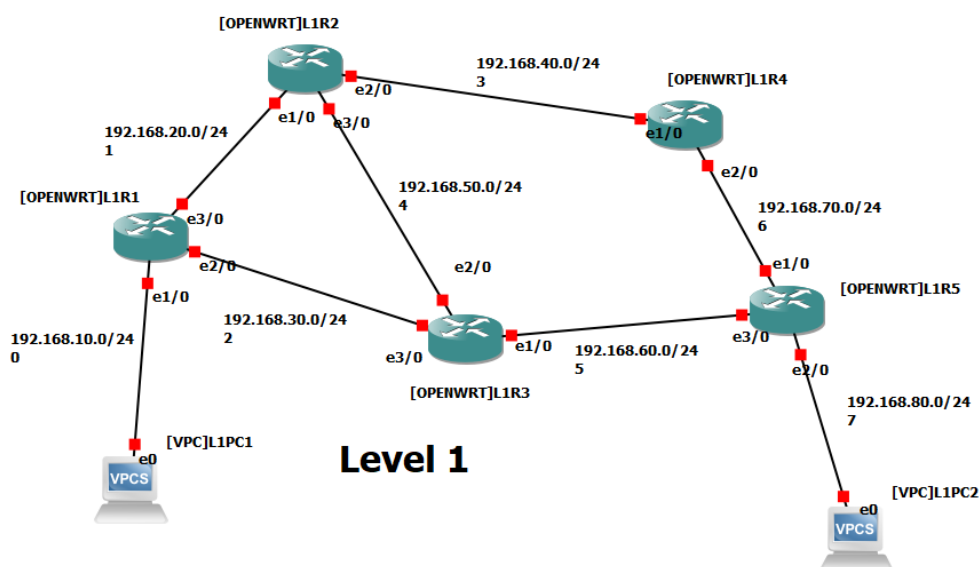


Figura 6.3: Red desplegada para el videojuego

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).

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[43].

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 pa-

ra 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 facilita 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 si estamos trabajando desde Windows (en Linux no es necesario), 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, $[OPENWRT]L1R1 \rightarrow [OPENWRT]L1R3 \rightarrow [OPENWRT]L1R5$.

Las redes que han sido marcadas en los enlaces de la figura 6.3 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.

Antes de pasar a la siguiente sección, es necesario aclarar que el prefijo “L1” del nombre de todos los nodos hace referencia a “Level 1”. Se da a entender por consiguiente que el proyecto parte de la idea de que los nodos de todos los niveles pasaran a estar en el mismo esquemático.

Despliegue

Los dispositivos se han interconectado tal y como se aprecia en la ilustración 6.3. 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 varios minutos para ser completamente funcionales en el PC desde el que se han realizado las pruebas. 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. 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.

6.2.3. 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](https://opengameart.org/), 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](https://opengameart.org/). 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 6.4), 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.

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.

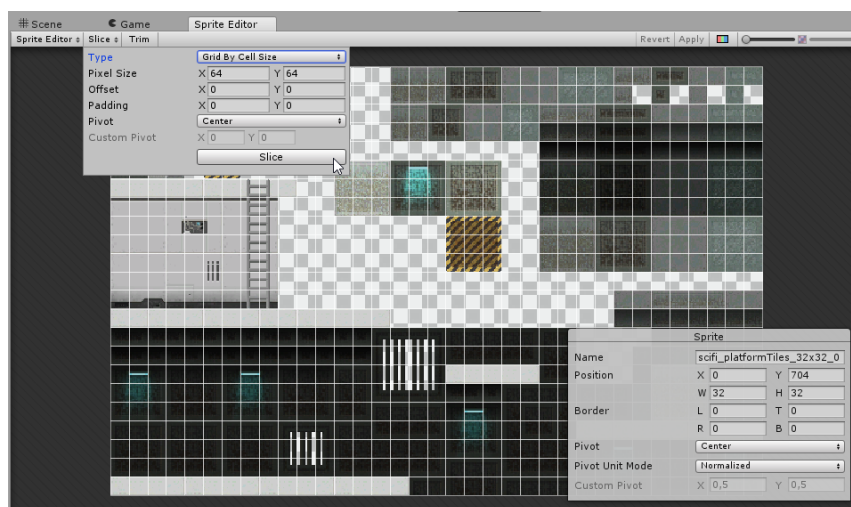


Figura 6.4: Editor de sprites de Unity

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 6.3. 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.

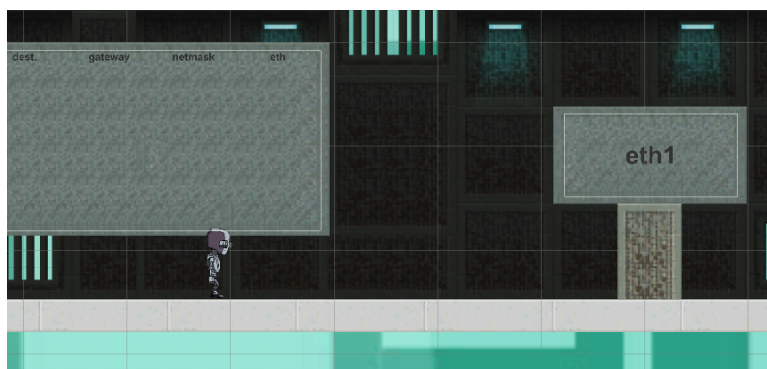


Figura 6.5: Plataforma del nivel del juego

En la figura 6.5 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[44]) y un par de elementos **collider** (definen la forma de un objeto para las colisiones físicas[45]).
- 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”[46]. Los tilemaps son, a su vez, una redcilla 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” (disparador) 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 6.6, donde pueden observarse las distintas plataformas que lo conforman.

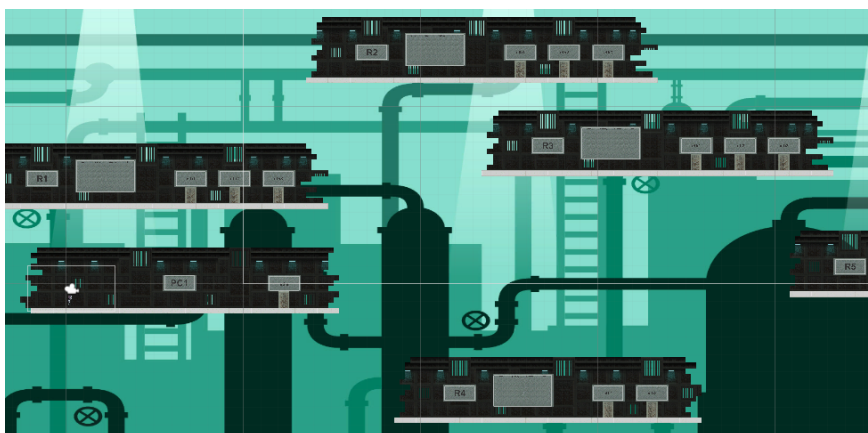


Figura 6.6: Escenario creado

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.

Para que estos puedan hacer uso de la librería que hemos desarrollado, hace falta depositar el `.dll` de la misma (y el `.xml` con la documentación que se generó junto a él) junto al de las dependencias que vimos en la subsección 6.1.7 en la carpeta “Assets” del proyecto de Unity. Esta carpeta aparece en todo proyecto y contiene, como su propio nombre indica, todos los elementos o assets de los que el juego se valdrá. Con el fin de que Unity procese correctamente la información del ensamblado, se requiere configurar las preferencias del proyecto para que este haga uso de `.NET 4.x` como motor de ejecutado de los scripts y de `.NET Standard 2.0` como nivel de compatibilidad de API.

- **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[47]. 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 6.11: 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)
            ;
    }

}

```

El 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.

- **L1Mapping**: esta clase define una serie de propiedades y métodos estáticos que son de ayuda para otras clases de la escena. Nació como un baúl donde guardar variables necesarias para ciertos elementos de la escena.
 - **L1Mapping.NodeNamesM1**: siguiendo la filosofía recién citada, guarda una cadena de **strings** con los nombres de los nodos que pueden encontrarse. Con “L1” nos referimos al nivel y con el “M1” al minijuego o fase dentro del mismo. Dado que no ha sido desarrollada más que una escena, esta sintaxis cae en desuso.
 - **L1Mapping.NodeRespawnM1**: para ir de una plataforma a otra del nivel el personaje es teletransportado. Esta propiedad guarda un mapeo entre cada una de las plataformas-nodos y la posición del mapa en la que el personaje debe reaparecer al acceder a cada una de ellas.

- `L1Mapping.DoorsOrderM1`: aleatoriza el orden de las puertas-interfaces de cada plataforma, logrando de esta manera que cambie a cada partida.
- `L1Mapping.RandomizeNets()`: aleatoriza la x de redes del tipo 192.168.x0.y; redes como las que usamos en nuestro nivel. Así, la configuración de la red varía también a cada ejecución del programa.

Código 6.12: `L1Mapping.RandomizeNets()`

```
public static ushort[] RandomizeNets(ushort
    numLinks)
{
    // Result list
    List<ushort> tempLinks = new List<ushort>(
        numLinks);
    // Initial list
    List<ushort> initialList = new List<ushort>(
        numLinks);
    // Random variable
    System.Random rnd = new System.Random();

    for (ushort i = 10; i <= numLinks * 10; i +=
        10) initialList.Add(i);
    for (ushort i = 0; i < numLinks; i++)
    {
        // Pick a random net number (10, 20, 30...)
        // from the list
        var randIdx = rnd.Next(0, initialList.Count
            );
        tempLinks.Add(initialList[randIdx]);
        initialList.RemoveAt(randIdx);
    }
    return tempLinks.ToArray();
}
```

- `L1M1Handler`: contiene la mayor parte de la lógica del nivel y, por tanto, ha de cargarse al comienzo de la escena. Inicializa todos los nodos del proyecto de GNS3 asociados al nivel, asigna variables con las referencias de cada uno de ellos y espera cuatro minutos a que todos ellos finalicen su arranque. Tras ello, configura los dispositivos del proyecto, comenzando por los PCs y continuando por los routers. El código del que se dispone a continuación muestra parte del proceso:

Código 6.13: `L1M1HandlerHelper.SetupRouters()` y `L1M1HandlerHelper.SetupRouters()`

```
public static void SetupRouters(OpenWRT[] Routers,
    ushort[] NetsPrefix) {
```

```

Task[] tasks = new Task[5]
{
    Task.Factory.StartNew(() => SetUpR1(Routers[0],
        NetsPrefix)),
    Task.Factory.StartNew(() => SetUpR2(Routers[1],
        NetsPrefix)),
    Task.Factory.StartNew(() => SetUpR3(Routers[2],
        NetsPrefix)),
    Task.Factory.StartNew(() => SetUpR4(Routers[3],
        NetsPrefix)),
    Task.Factory.StartNew(() => SetUpR5(Routers[4],
        NetsPrefix))
};
Task.WaitAll(tasks);
}

private static void SetUpR1(OpenWRT R, ushort[]
NetsPrefix) {
    // Net interfaces
    R.ActivateInterface(IP: $"192.168.{NetsPrefix[0]}.1",
        interfaceNumber: 1, netmask: "255.255.255.0");
    ;
    R.ActivateInterface(IP: $"192.168.{NetsPrefix[2]}.2",
        interfaceNumber: 2, netmask: "255.255.255.0");
    ;
    R.ActivateInterface(IP: $"192.168.{NetsPrefix[1]}.3",
        interfaceNumber: 3, netmask: "255.255.255.0");
    ;
    R.DisableFirewall();
    // Routes
    R.SetRoute(destination: $"192.168.{NetsPrefix[7]}.0",
        gateway: $"192.168.{NetsPrefix[2]}.3",
        netmask: "255.255.255.0");
    R.SetRoute(destination: $"192.168.{NetsPrefix[5]}.0",
        gateway: $"192.168.{NetsPrefix[2]}.3",
        netmask: "255.255.255.0");
    R.SetRoute(destination: $"192.168.{NetsPrefix[3]}.0",
        gateway: $"192.168.{NetsPrefix[1]}.1",
        netmask: "255.255.255.0");
    R.SetRoute(destination: $"192.168.{NetsPrefix[4]}.0",
        gateway: $"192.168.{NetsPrefix[2]}.3",
        netmask: "255.255.255.0");
    R.SetRoute(destination: $"192.168.{NetsPrefix[6]}.0",
        gateway: $"192.168.{NetsPrefix[1]}.1",
        netmask: "255.255.255.0");
}

```

Los dos métodos mostrados forman parte de `L1M1HandlerHelper`, una clase contenida dentro de `L1M1Handler` con algunas funciones que esta necesita. Tal como puede intuirse en las líneas de `SetupRouters()`, se ha paralelizado la configuración de los routers, reduciendo drásticamente el tiempo de inicio del nivel. Esto se lleva a cabo mediante

la clase predefinida de C# **Task**, que permite la definición de tareas ejecutadas en hilos independientes.

Existe una tarea para la configuración de cada router, llevada a cabo con funciones como **SetUpR1()**.

Finalmente, **L1M1Handler** también se encarga de cosas como el dibujo de las tablas de enturamiento en el nivel y la creación de un contador ascendente que se muestra continuamente en pantalla. La finalidad de este último es la de incentivar al jugador a terminar el mapa en el menor tiempo posible. De haberse desarrollado más el juego, habría cobrado más sentido, con elementos como bases de datos donde guardar las puntuaciones de cada partida.

- **DoorOrder**: tiene como misión asignar el cartel que habrá encima de cada puerta así como determinar cuál será la posición en la que reaparezca el protagonista cuando decida atravesar una de ellas.

El script es insertado en unos **GameObjects** utilizados para representar las puertas en la escena, permitiéndole a aquel tomar parámetros personalizados de cada una de ellas. Así, conociendo el número de la puerta y la plataforma-nodo en la que reside, mediante **L1Mapping** puede averiguar fácilmente cuál es la interfaz asociada a ella y cuál es la posición que el personaje obtendrá una vez decida atravesarla.

Como el cartel ubicado encima de cada puerta marcará la IP asociada a la interfaz del nodo correspondiente, es necesario buscar el modo de recibirla. La forma de hacerlo podría “trampearse” a través de alguno de los mapeos de **L1Mapping**, pero se ha decidido llevarlo a cabo mediante el método **OpenWrt.GetIPByInterface()**, que extrae la información a través de una consulta al router. La problemática asociada a esta solución es el tiempo que requiere, pues ha de hacerse por cada puerta y no es en absoluto un proceso automático.

- Otras clases de menor calibre que no nos detendremos a explicar son:
 - **EscToExit**: permite que al pulsar “Esc” se salga del juego.
 - **ExistLevel**: cierra el juego al llegar a la plataforma-PC final.
 - **FallDown**: devuelve al personaje a una zona segura si se cae de alguna plataforma.
 - **Teletransport**: desplaza la posición del personaje hasta unas ciertas coordenadas del escenario.
 - **SceneLoader**: se incrusta en la primera escena y carga desde ella la segunda. El script está tomado directamente de otra persona, a la que se da crédito en el propio código.

Capítulo 7

Pruebas

En el penúltimo capítulo del documento se mostrarán los resultados del proyecto; se hará un análisis del establecimiento del juego y del rendimiento del simulador que lo soporta.

7.1. Descripción de las pruebas

Una vez asentado todo lo que vimos en el capítulo 6, vamos a probar su funcionamiento así como su rendimiento. Hemos decidido llevar a cabo las pruebas en dos entornos diferentes, para ponderar ventajas e inconvenientes de cada uno de ellos. Estos se describen a continuación:

- El **caso 1** es sobre el que todo el desarrollo del proyecto ha sido ejecutado. El entorno consiste en una máquina física con Windows 10 desde donde el juego es ejecutado y GNS3 debe estar corriendo. Como se trata de un sistema operativo Windows, nodos como los de OpenWRT necesitan ser montados sobre un GNS3 VM, lo cual implica hacer uso de un hipervisor (aquí usaremos Virtualbox, pues aunque su rendimiento es inferior al de VMware, es open source) donde instalarlo. El proceso de despliegue es entonces algo tedioso, ya que requiere de una instalación de GNS3, una configuración apropiada del mismo y una máquina virtual con GNS3 VM.
- El **caso 2** busca acabar con ese proceso. La idea aquí es usar una sola máquina virtual como contenedor de todo lo que el juego necesita del simulador. Para ello, se ha preparado una máquina virtual con Xubuntu (recaltar que el adaptador de red está establecido como “puente”, ya que así que la máquina toma una dirección de red diferente de aquella de la máquina huésped) y sobre él se ha instalado GNS3; se ha configurado este, se ha creado el proyecto de la figura 6.3 y finalmente

se ha exportado la máquina en formato *.ova*. De esta forma ganamos en portabilidad. Además, puede demostrar que la API funciona incluso cuando el GNS3 al que llama es remoto.

La máquina virtual tiene asignados dos núcleos del procesador.

En ambos casos, se llevarán a cabo varias pruebas sobre el proyecto de la figura 6.3.

- La primera de ellas observará el consumo de RAM que GNS3 toma, tanto cuando el proyecto está parado como en acción. Para ello son usadas dos herramientas de Microsoft: Process Explorer y RamMap. Usaríamos únicamente el primero de ellos si no fuera porque los hipervisores utilizan algo llamado “memoria bloqueada por el controlador” (*driver locked memory*) que los monitores de recursos normales no muestran. La memoria bloqueada por el controlador aparece cuando un controlador modo-kernel evita que las páginas de memoria sean cambiadas al archivo de paginación. Es a través de este mecanismo que el hipervisor varía la cantidad de memoria disponible para un huésped cuando la memoria dinámica está activada [48].
- La segunda medirá cuánto tiempo tarda cada router OpenWRT en iniciarse de forma aislada (se arranca él solo, no el proyecto completo). Para ello, se empieza a cronometrar desde que se pulsa el botón de inicio del nodo hasta que en el terminal del router aparece el mensaje *br-lan: port 1(eth0) entered forwarding state*, a partir del cual el nodo comienza a ser plenamente funcional. También se testeará el caso análogo para cuando se ejecuta el proyecto completo. En esta situación, se mostrará el tiempo transcurrido hasta que todos los routers sean correctamente inicializados.
- La última medirá el tiempo que Unity necesita para cargar la escena tomando los datos del simulador. No se contará aquí el la espera establecida manualmente para que el juego no trabaje hasta que los aparatos estén disponibles. Extraeremos tres datos:
 1. Tiempo de establecimiento de los VPCs: cuánto tiempo se necesita para que su configuración sea establecida en GNS3.
 2. Tiempo de establecimiento de los routers: cuánto tiempo se necesita para que su configuración sea establecida en GNS3. Recordar que esta tarea fue paralelizada de modo que se establecían todos “a la vez” y no secuencialmente.
 3. Tiempo de establecimiento de las tablas de encaminamiento: cuánto tiempo necesita Unity para pedir los datos a los routers (tablas de encaminamiento, IPs asociadas a cierta interfaz) con el fin de rellenar los carteles de la escena.

Llevaremos a cabo esta prueba de forma automatizada sobre el juego compilado, insertando una instancia `StopWatch` en la clase `L1M1Handler` y exportando los distintos lapsos de tiempo en un fichero de texto que pueda ser consultado tras su escritura.

7.2. Caso 1

7.2.1. Consumo de RAM

La memoria bloqueada en el PC antes de que GNS3 y la máquina virtual de GNS3 VM sean abiertos es de 13652KB, los cuales serán descontados del total.

- Cuando el proyecto es abierto pero **ningún nodo es iniciado**, GNS3 aloja 131,7MB mientras que GNS3 VM 28,3125MB. Total: **160,01MB**.
- **Arrancado** el proyecto, GNS3 152,1MB y GNS3 VM se mantiene igual. Total: **180,41MB**.

7.2.2. Tiempo de arranque de los routers

La siguiente tabla recoge el tiempo de arranque individual de cada nodo y la media del total.

R1	R2	R3	R4	R5	Media
52,28s	52,38s	47,13s	51,26s	51,73s	50,956

Si ahora probamos a arrancar el proyecto completo (que incluye dos VPCs además de los routers), el resultado se dispara, necesitando de nada menos que **4min15s** para que todos los nodos sean completamente cargados. Es más de cuatro veces la media de lo que necesita cuando son ejecutados individualmente.

7.2.3. Tiempo de establecimiento de la escena

La tabla 7.2.3 recoge los datos obtenidos tras la prueba:

Establecimiento VPCs	Establecimiento routers	Establecimiento tablas de encaminamiento
0,017s	42,255s	62,299s

7.3. Caso 2

7.3.1. Consumo de RAM

De nuevo, la memoria bloqueada en el PC antes del experimento es de 13652KB.

- Cuando **está parado** el proyecto, la máquina virtual que contiene Xubuntu y donde está GNS3 instalado ocupa **261,61MB**.
- Cuando se **inicia** todo el proyecto, la RAM usada pasa a ser de **790,61MB**.

7.3.2. Tiempo de arranque de los routers

La siguiente tabla recoge el tiempo de arranque individual de cada nodo y la media del total.

R1	R2	R3	R4	R5	Media
60,26s	57,96s	58,21s	58,44s	57,95s	58,564

Como es de esperar, el tiempo que tardan todos los nodos en ser completamente inicializados también se ve incrementado. Hasta los **5min16s** concretamente.

7.3.3. Tiempo de establecimiento de la escena

La tabla 7.3.3 recoge los datos obtenidos tras la prueba:

Establecimiento VPCs	Establecimiento routers	Establecimiento tablas de encaminamiento
0,017s	46,246s	66,294s

7.4. Conclusiones

Como era de esperar, el caso 2, al estar basado íntegramente en una máquina virtual, ofrece peores resultados que el caso 1, ya que esta cuenta con los recursos de la máquina física de forma muy limitada y controlada.

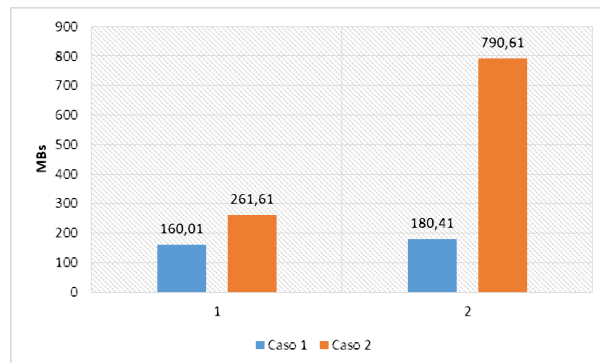


Figura 7.1: Comparación consumo de RAM

La figura 7.1 evidencia la diferencia de consumo de RAM entre ambas opciones. Mientras que la primera apenas llega a los 200MBs, la segunda la cuadriplica. En consecuencia, será necesario contar con un PC de ciertas prestaciones para ser capaz de trabajar con redes grandes de usar este método.

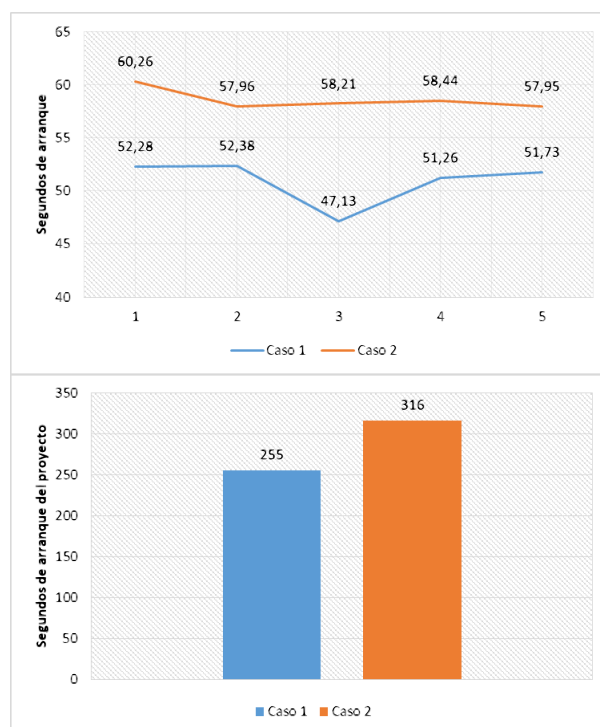


Figura 7.2: Tiempos de arranque de los nodos de forma individual (arriba) y en grupo (abajo)

El resultado también es negativo para el tiempo de arranque de los routers. Si bien en ninguno de los dos casos la activación es automática como veníamos anunciando, el caso 2 sigue quedando detrás según se observa en la gráfica 7.2.

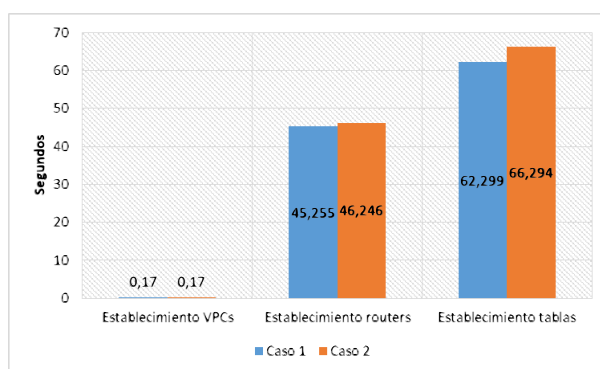


Figura 7.3: Tiempos de arranque de las partes del juego relacionadas con el simulador

Finalmente, la figura 7.3 muestra que las diferencias a la hora de interactuar con el juego no son demasiado grandes entre los dos casos. El caso 1 es ligeramente más rápido que el segundo.

Aunque es evidente la derrota en rendimiento de la máquina virtual, las ventajas con ella son claras como se explicó al comienzo del capítulo. El hecho de aislar toda la configuración del simulador en una sola máquina virtual hace su uso extremadamente portable y sencillo. Problemas como su mayor consumo de RAM no debe serlo a día de hoy, con sistemas que cuentan fácilmente con 8GBs de RAM. Además, la sola instalación de GNS3 es ya de por sí poco recomendable para equipos de pocas prestaciones [15]. Yendo aún a más: si el proyecto se encuentra inicialmente activo y configurado cuando el juego fuera a hacer uso de él, el caso 2 no apenas se ve perjudicado frente al segundo en cuanto a configuración del juego e interacción con este.

Consideramos entonces que es la opción correcta a seguir, ya que firmemente pensamos que las ventajas que ofrece las hacen ideales para el proyecto (acercar el juego al mayor número de personas) frente a los inconvenientes que es evidente que presenta.

Capítulo 8

Conclusiones

Antes de finalizar el documento, recordaremos los puntos claves de lo que se pudo leer en él y haremos una valoración global del proyecto.

8.1. Trabajo realizado

Durante el presente documento hemos atendido principalmente a dos asuntos.

El primero de ellos describe con todo detalle la forma de hacer interactiva una red virtualizada a través de una herramienta externa a ella. La red virtualizada es generada mediante el simulador GNS3, mientras que la herramienta se materializó en una librería para el lenguaje C#. Tal biblioteca nos permite desde encender y apagar dispositivos por la red, pasando por crear nuevos enlaces entre ellos, hasta controlarlos desde dentro tal y como necesitemos. Las posibilidades que brinda son amplias.

El segundo punto refiere directamente a la creación de un juego haciendo uso de la utilidad creada en el apartado anterior. Gracias a ella, se consiguió crear un pequeño “puzzle” cuyas indicaciones eran extraídas directamente desde la red virtual. El puzzle pone a prueba los conocimientos de enrutamiento del jugador.

8.2. Valoración del resultado

La API desarrollada puede suponer una herramienta de increíble utilidad para aplicaciones de toda índole en el mundo de la telemática. Al haber sido cuidada de forma tan detallada, su utilización e incluso modificación posterior no posee complejidad alguna. Consideramos entonces que, en cuanto cómo de reutilizable puede llegar a ser, es un éxito. Nos hemos encargado

de que exista información suficiente para aquel que pretenda retomarla mediante documentación tanto en el código como en el repositorio donde este se hospeda.

Existen problemas que aún no hemos podido resolver con éxito en la librería. Así por ejemplo, establecida una conexión con los nodos, cada envío y recepción de mensajes lleva un tiempo. La mayor parte de las veces son problemas, no directamente de la API, sino del propio simulador. Pongamos por caso uno de los más problemáticos: el arranque de cada dispositivo. Si el aparato en concreto es pesado, GNS3 necesita de un tiempo del orden de minutos para que sea satisfactoriamente iniciado, lo que implica un retraso para cualquiera que sea la aplicación desde la que se esté lanzando la librería. Además, esta aún no ha encontrado el modo de automatizar la comprobación de la operatividad de un nodo para comenzar a trabajar con él. Es necesario entonces escribir una sentencia de pausa en la aplicación, lo cual es rudimentario y poco preciso y por ende implica fallos en potencia.

Pese a todo ello la librería es plenamente funcional. Tiene limitaciones como el no poder generar proyectos de GNS3 desde cero o su dependencia con aquel programa, que necesita ser inicializado previamente de forma manual (no se ha automatizado su arranque) para poder trabajar con él.

Posiblemente el verdadero punto débil del proyecto haya sido el videojuego. Y es que aunque al comienzo se pretendía llevar a cabo un juego completo que explotará la API al máximo y que fuera capaz de explicar varias asignaturas de la telemática, finalmente se optó por crear algo mucho más pequeño. El desarrollo de un videojuego completo es una tarea de gran complejidad que requiere de un número de horas mucho mayor del que era viable dedicar a este proyecto. Para adaptarse a las circunstancias, la decisión tomada fue la de crear un solo nivel que demostrase que la interacción con la biblioteca es funcional. El resultado pudo verse en el capítulo 7.

Si bien efectivamente se comprobó la viabilidad de la relación $\text{GNS3} \leftrightarrow \text{API} \leftrightarrow \text{Juego}$, quizá no fuera explotada lo suficiente. Muchas de las características insertadas en la librería han quedado excluidas del juego final pues no son útiles para el mismo. Añadir nuevos niveles habría garantizado su inclusión en el mismo, pero ello también implicaba una planificación más detenida y un mayor desembolso temporal inviable.

En cualquiera de los casos, como ha quedado demostrado en el documento, el proyecto ha estado centrado en la interacción entre los distintos elementos de nuestro problema.

8.3. Experiencia personal

El desarrollo de este proyecto ha supuesto un aprendizaje continuo para mí. He descubierto qué son los simuladores de redes, me he introducido en el mundo de la programación de videojuegos, he aprendido a C# y a trabajar en el entorno .NET... y realmente un largo número de etcéteras más. Ha sido realmente una experiencia plena porque todo ese aprendizaje provenía de uno mismo: de las constantes batallas contra los problemas aparecidos y de la habilidad para resolverlos con los medios dados.

Por un lado me siento plenamente satisfecho con el trabajo realizado, pues he abierto un camino sobre el que firmemente pienso que es importante seguir trabajando. Me apena saber que el videojuego que tenía en mente cuando comencé el proyecto no haya podido llevarse a cabo. Mi nula experiencia con motores de juegos me ha ralentizado enormemente a la hora de crear. Esto se traduce en grandes cantidades de tiempo invertidas en pasos elementales y pequeños, reduciendo considerablemente el número de horas para el desarrollo de algo más grande.

8.4. Trabajos futuros

Tal y como se ha repetido en varias ocasiones, la API ha sido creada de forma que cualquier nuevo desarrollador que pretende hacer uso de ella, lo tenga fácil para ello. Así, proponemos que más adelante el proyecto sea retomado.

Como el presente trabajo se ha propuesto y ha logrado crear la interacción entre el videojuego y la red virtualizada, la persona que encargada de reiniciarlo tendría la responsabilidad de completar lo relativo al videojuego. Este traspaso de testigo consistiría en construir un videojuego de ciertas dimensiones que ponga en práctica lo teorizado durante el documento; algo como lo que podíamos ver en la sección 5.2.2. Disponible la herramienta y las indicaciones necesarias para hacerlo, solo queda ponerlas en práctica.

Bibliografía

- [1] Geraldine O'Neill and Tim McMahon. Student-centred learning: What does it mean for students and lectures? *AISHE*, 1:27–36, 2005. Disponible en <http://eprints.teachingandlearning.ie/3345/1/0%27Neill%20and%20McMahon%202005.pdf>.
- [2] Joy Crosby R.M. Harden. The good teacher is more than a lecturer - the twelve roles of the teacher. *Medical teacher*, 22(4):334–347, 2005. Disponible en <http://dx.doi.org/10.1080/014215900409429>.
- [3] Jennifer Tiede y Silke Grafe. Triseum game-based learning validation study. Technical report, European Schoolnet, Junio 2018. Disponible en http://storage.eun.org/eun-form-submission/1226/Triseum_Pilot_report-1.pdf.
- [4] Angela McFarlane John Kirriemuir. Use of computer and video games in the classroom. Disponible en <http://www.digra.org/wp-content/uploads/digital-library/05150.28025.pdf>.
- [5] Adriana D'Alba Yun-Jo An, Linda Haynes and Frances Chumney. Using educational computer games in the classroom: Science teachers' experiences, attitudes, perceptions, concerns, and support needs. Disponible en <http://www.citejournal.org/volume-16/issue-4-16/science/using-educational-computer-games-in-the-classroom-science-teachers-experiences-attitudes-perceptions-concerns-and-support-needs>.
- [6] Shannon Kennedy-Clark and Kate Thompson. What do students learn when collaboratively using a computer game in the study of historical disease epidemics, and why? *Sage*, 6(6):513–537, 2011. Disponible en <http://journals.sagepub.com/doi/abs/10.1177/1555412011431361>.
- [7] Iván Lobo y Noelia Núñez. Aprender ciencia jugando en laboratorios virtuales. Disponible en <https://elfuturoesapasionante.elpais.com/labster-aprender-ciencia-jugando-laboratorios-virtuales/>, Noviembre 2017.

- [8] Real Academia de Ingeniería. Diccionario español de ingeniería, 2018. Disponible en <http://diccionario.raing.es/es/lema/telem%C3%A1tica-0>.
- [9] Wikipedia. Network simulation — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Network%20simulation&oldid=847300944>, 2018. [Online; accessed 31-August-2018].
- [10] Zak Cole. Network simulation or emulation? Disponible en <https://www.networkworld.com/article/3227076/lan-wan/network-simulation-or-emulation.html>, Septiembre 2017.
- [11] Mininet overview. Disponible en <http://mininet.org/overview/>.
- [12] *CORE Documentation*, 4.8 edition, Junio 2015. Disponible en https://downloads.pf.itd.nrl.navy.mil/docs/core/core_manual.pdf.
- [13] *CORE IPC API*, 1.23 edition. Disponible en https://downloads.pf.itd.nrl.navy.mil/docs/core/core_api.pdf.
- [14] Jason C. Neumann. *The Book of GNS3*. no starch press, 1st edition, 2015.
- [15] What is gns3? Disponible en <https://gns3.com/software>.
- [16] Joseph Hocking. *Unity in action*. Manning, 2000.
- [17] Wikipedia. List of Unreal Engine games — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=List%20of%20Unreal%20Engine%20games&oldid=857258942>, 2018. [Online; accedido el 01-septiembre-2018].
- [18] Frequently asked questions (faq). Disponible en <https://www.unrealengine.com/en-US/faq>.
- [19] Unreal engine features. Disponible en <https://www.unrealengine.com/en-US/features>.
- [20] Alex Wawro. For tim sweeney, advancing epic means racing into ar and vr. Disponible en http://www.gamasutra.com/view/news/292573/For_Tim_Sweeney_advancing_Epic_means_racing_into_AR_and_VR.php, Marzo 2017.
- [21] What is the easiest to learn between unity and unreal engine? Disponible en <https://www.quora.com/What-is-the-easiest-to-learn-between-Unity-and-Unreal-Engine>.

- [22] The top 10 video game engines. Disponible en <https://www.gamedesigning.org/career/video-game-engines/>.
- [23] Gamemaker showcase. Disponible en <https://www.yoyogames.com/showcase>.
- [24] Get gamemaker. Disponible en <https://www.yoyogames.com/get>.
- [25] Features. Disponible en <https://www.yoyogames.com/gamemaker/features>.
- [26] *GameMaker Language Overview*. Disponible en https://docs2.yoyogames.com/index.html?page=source%2F_build%2F3_scripting%2F3_gml_overview%2Findex.html.
- [27] Michael Larabel. Godot 3.0 open-source game engine released. Disponible en https://www.phoronix.com/scan.php?page=news_item&px=Godot-3.0-Released, Enero 2018.
- [28] Features. Disponible en <https://godotengine.org/features>.
- [29] Licence. Disponible en <https://godotengine.org/license>.
- [30] Qemu. Disponible en <https://www.qemu.org/>.
- [31] Julien Duponchelle y Jeremy Grossmann. *GNS3 structure*. Disponible en <https://gns3-server.readthedocs.io/en/latest/general.html>.
- [32] Introducing json. Disponible en <https://www.json.org/>.
- [33] *What Is The GNS3 VM?*, Septiembre 2015. Disponible en <https://gns3.com/news/article/what-is-the-gns3-vm>.
- [34] 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].
- [35] Google maps platform. Disponible en <https://cloud.google.com/maps-platform/?hl=es>.
- [36] Pavol Návrát. 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>.
- [37] 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.

- [38] Esolang, the esoteric programming languages wiki. Disponible en https://esolangs.org/wiki/Hello_world_program_in_esoteric_languages.
- [39] Ignacio Roldán Etcheverry. Introducing c# in godot. Disponible en <https://godotengine.org/article/introducing-csharp-godot>, Octubre 2017.
- [40] Armina Mkhitarian. 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-ccf2682>, Octubre 2017.
- [41] Joseph Albahari and Ben Albahari. *C# 7.0 in a Nutshell*. O'Reilly Media, 7th edition, 2018.
- [42] Microsoft. *Clase WebClient*, Octubre 2016. Disponible en [https://msdn.microsoft.com/es-es/library/system.net.webclient\(v=vs.110\).aspx](https://msdn.microsoft.com/es-es/library/system.net.webclient(v=vs.110).aspx).
- [43] About openwrt. Disponible en <https://wiki.openwrt.org/about/start>.
- [44] Unity. *Rigidbody 2D*. Disponible en <https://docs.unity3d.com/Manual/class-Rigidbody2D.html>.
- [45] Unity. *Colliders*. Disponible en <https://docs.unity3d.com/es/current/Manual/CollidersOverview.html>.
- [46] Unity. *Tile*. Disponible en <https://docs.unity3d.com/Manual/Tilemap-ScriptableTiles-Tile.html>.
- [47] Jon Skeet. *C# in Depth*. Manning, 3rd edition, 2013. Disponible en <http://csharpindepth.com/Articles/General/Singleton.aspx>.
- [48] Doug Luxem. Driver locked memory on hyper-v guests. Disponible en <http://blog.luxem.org/2013/05/driver-locked-memory-on-hyper-v-guests.html>, Mayo 2013.
- [49] 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>.
- [50] Microsoft. *An introduction to NuGet*, Enero 2018. Disponible en <https://docs.microsoft.com/es-es/nuget/what-is-nuget>.

