

**Herramienta para la sencilla creación de servidores eficientes en Node.js
Implementada en el juego “Save The King”**

Javier Iregui Conde
1 de Septiembre de 2023

ESNE - Escuela Universitaria de Diseño, Innovación y Tecnología
Diseño y Desarrollo de Videojuegos - Programación
Curso 2022-2023



Índice de figuras	3
Resumen	4
Abstract	4
1. Introducción	5
Save the King	5
WebRTC Game Server	5
2. Objetivos generales	7
3. El estado de la cuestión	7
Arquitecturas de comunicación	8
Transporte: TCP y UDP	10
Comunicación en tiempo real: Web Sockets y WebRTC	12
Seguridad: El modelo Tor	15
Alquiler de servidores: Infraestructura como servicio	16
Soluciones disponibles para el desarrollo de servidores	18
4. Metodología	18
5. Desarrollo del proyecto	19
Roles de programación	19
Análisis	19
Relación de tareas	20
6. Resultados y conclusiones	23
Resultados	23
Conclusiones	24
7. Post mortem y líneas de futuro	24
8. Anexos	26
9. Bibliografía	27

Índice de figuras

Figura 1: Arquitecturas de comunicación	9
Figura 2: Modelo OSI	10
Figura 3: Modelo TCP/IP	11
Figura 4: Transferencia de datos HTTP vs. Socket.io	13
Figura 5: Diagrama Tor Net	15
Figura 6: Precios Cloud Computing	17
Figura 7: Modelo de datos de la sala de juego	20
Figura 8: Conexión WebRTC realizada	22
Figura 9: Uso de recursos WebRTC Game Server	24

Resumen

Un enorme porcentaje de los videojuegos que se desarrollan a día de hoy son tanto multijugador como indie, sin embargo, esta categoría incluye proyectos con grandes diferencias en múltiples aspectos como los recursos disponibles. Además, gracias al avance tecnológico, el mundo del desarrollo de videojuegos está abierto a cualquier persona con acceso a un portátil, aún así, en muchos casos, este tipo de desarrolladores no tienen los recursos para contratar servidores en servicios como AWS o, simplemente, no consideran que valga la pena para algunos de sus proyectos. Con este proyecto se pretende crear una herramienta para desarrollar, de una manera sencilla, aplicaciones de servidor personalizadas a cada proyecto que, además, liberen carga del servidor permitiendo utilizar máquinas con menos recursos o ahorrar en servicios de alquiler de servidores. Como resultado del proyecto, se publicará a GitHub un repositorio con la intención de que sea clonado para ser utilizado como base en otros proyectos de Node.js. Este proyecto funcionará gracias a WebRTC, lo que permite que, al conectarse al juego en cuestión, el servidor asigne una sala en la que uno de los clientes estará manejando el tráfico de la partida, sustituyendo así al servidor central y simplificando sus funciones. Gracias a este proyecto, otros desarrolladores podrán agilizar la implementación de un sistema multijugador por salas basado en tecnologías web y con un coste reducido, o incluso nulo, debido a que gran parte de los servicios de alquiler de servidores ofrecen servicio gratis dependiendo del uso de los mismos.

servidor web, videojuegos, node.js, express, web socket, socket.io, webrtc

Abstract

A significant percentage of today's video games are both multiplayer and indie. However, this category encompasses projects with significant differences in various aspects such as available resources. Additionally, thanks to technological advancements, the world of game development is now open to anyone with access to a laptop. Nevertheless, in many cases, these developers may not have the resources to rent servers on services like AWS or they simply do not consider it worthwhile for some of their games. This project aims to create a tool that simplifies the development of custom server applications suitable for any project. This tool is also meant to reduce the server's work load, enabling the use of machines with fewer resources or saving on server rental fees. As a result of the project, a GitHub repository will be published with the intention that it can be cloned and used as a foundation for other Node.js projects. This project will function thanks to WebRTC, which allows the server, upon connecting to the game in question, to allocate a room in which one of the clients will manage the game traffic thereby replacing the central server and simplifying its functions. Thanks to this project, other developers will be able to streamline the implementation of a room-based multiplayer system based on web technologies with reduced or even zero costs, as many server rental services offer free usage depending on their usage.

web server, video games, node.js, express, web socket, socket.io, webrtc

1. Introducción

Save the King

Parte del proyecto era realizar un videojuego de manera grupal que incluyese las tecnologías desarrolladas sobre las investigaciones individuales. En este caso, una de las investigaciones se dirigió a la inteligencia artificial y, la otra, al back end y el multijugador. Ambas investigaciones fueron componentes imprescindibles en el desarrollo de la parte grupal: el juego web “Save the King”.

Save The King es un videojuego web multijugador para navegadores basado en una especie de “pilla pilla” por equipos. Debido a su naturaleza estratégica y por su estética, podría compararse a juegos como el “Among Us” por lo que su público objetivo es similar, es decir, jugadores jóvenes que jugarán especialmente en situaciones específicas cuando se encuentren con amigos.

Durante la partida, dos equipos se enfrentarán entre ellos para cazar al rey del equipo opuesto. La mecánica principal será el movimiento por diferentes mapas dependiendo de la partida, que estarán llenos de escondites y cada uno tendrá una estética. Además, existirá un sprint que se recargará mientras se anda y que deberá ser administrado para poder alcanzar a los enemigos.

Uno de los aspectos más importantes del juego es la estrategia, ya que cada jugador del equipo tendrá un papel dentro del mismo. Los roles dependerán del color del jugador, que será asignado de manera aleatoria al comienzo de la partida y, dependiendo del mismo, el jugador podrá ser el rey, es decir, el color blanco y el personaje que debe sobrevivir para no perder la partida, o un color primario o secundario. Ciertos colores podrán eliminar a otros, pero también habrá jugadores que puedan eliminarlos. Esto se mostrará en la interfaz del jugador. Los colores del equipo opuesto aparecerán ocultos al jugador, y solo se mostrarán cuando el enemigo en cuestión haya entrado en combate al menos una vez al ser pillado. De esta manera se crearán partidas rápidas y dinámicas en las que se debe correr por el mapa para pillar a enemigos, eliminarlos, descubrir sus colores, encontrar al rey y, por último, acabar con él.

El juego será gratuito, aunque en un futuro podría añadirse una tienda para obtener pequeñas modificaciones estéticas para el jugador y acceso a nuevos mapas mediante microtransacciones.

Una de las investigaciones individuales se centrará en el back end, los servidores, el sistema de salas de juego y la optimización de la comunicación entre jugadores. De esta manera, se creará la base de código que permitirá la funcionalidad principal del juego. Por otro lado, la inteligencia artificial, es decir, el otro tema de investigación individual, cobrará importancia debido a que las partidas deben tener un número mínimo de usuarios para que las mecánicas de juego tengan sentido. Además, aunque sea posible jugar con 6 jugadores como mínimo, la situación perfecta para el juego es de 14 por partida. No siempre será posible conseguir este número de jugadores en un tiempo de espera razonable, por lo que la partida comenzará con los usuarios disponibles y el resto de posiciones serán cubiertas por personajes controlados por la máquina. La segunda investigación individual se centrará en crear un sistema que controle a esos jugadores automáticos de manera adecuada.

WebRTC Game Server

Durante el desarrollo del videojuego grupal “Save the King” se realizaron tareas de forma individual. Entre ellas se encuentran el diseño de las mecánicas del juego, la conversión de las ideas de niveles y personajes a sus diseños digitales finales y, especialmente, el back end. Esta memoria y la investigación que describe estarán centradas en esa última parte de la aportación individual al proyecto “Save the King”: el back end.

Esta investigación se ha realizado ya que, a día de hoy, el mundo de los videojuegos cooperativos, multijugador u online domina gran parte de la industria. Una industria que, no solamente incluye los productos ya disponibles, sino que también goza de una gran comunidad de desarrolladores. Poco a poco, el desarrollo de videojuegos está sobrepasando su clásica frontera que lo limitaba a las grandes producciones y los conocidos como “AAA” o “triple A”. Gracias al avance tecnológico y la popularización de los videojuegos como pasatiempo, el desarrollo de los mismos también ha adquirido una cierta posición de “hobby”. Actualmente, no se necesitan máquinas tremendamente potentes ni conocimientos tan especializados, como podría ser el caso hace solamente unas décadas, para crear un pequeño proyecto funcional o incluso un producto viable con cierto nivel de popularidad. Sin embargo, la creciente y ya mencionada presencia del multijugador en la industria puede representar una frontera diferente a la expansión de la comunidad de desarrolladores indie.

Los videojuegos multijugador requieren de conocimientos sobre desarrollo back end, redes, bases de datos, y otras tecnologías, dependiendo de las necesidades específicas del proyecto. Además, en la gran mayoría de los casos se necesita una infraestructura de servidores que deben ser configurados y mantenidos para que puedan manejar el tráfico de información que ocurre entre los jugadores durante la partida.

Ambos componentes del problema sobre la accesibilidad a la industria del videojuego multijugador, es decir, los conocimientos y la infraestructura necesaria, han recibido numerosas soluciones a lo largo de los años. En principio, existen varias herramientas que simplifican la creación de los servidores, reduciendo la carga de trabajo que debe asumir el desarrollador. De todas formas, estas herramientas suelen ser demasiado sencillas, lo que las convierte en poco adaptables a las necesidades del proyecto en el que son empleadas o, en su defecto, y debido a su complejidad, siguen requiriendo un estudio previo de las mismas y, en ciertos casos, conocimientos más profundos sobre redes. Por otro lado, la infraestructura necesaria se puede construir de varias maneras.

Existe un tipo de arquitectura llamada “Peer to Peer” o “P2P” que se basa en la comunicación directa entre dos jugadores y permite prescindir de los servidores físicos, sin embargo, este tipo de comunicación no es adecuada para todo tipo de juegos y mecánicas y, por lo general, solía ser utilizada hace años para juegos basados en turnos. Actualmente, es más común la comunicación mediante un servidor. A esto se le llama modelo cliente-servidor. Estos servidores pueden obtenerse principalmente de dos maneras: la propiedad de los mismos, la cual conlleva su mantenimiento y costes de adquisición, o el alquiler. El precio de un servidor puede llegar a ser muy elevado dependiendo de sus componentes y su potencia, es decir, su capacidad de manejar el tráfico de los clientes. El momento en el que hay suficientes jugadores o el proyecto es demasiado amplio, un pequeño servidor de bajo coste no puede cubrir las necesidades de los jugadores. Es por esto que lo más común es alquilar servidores en servicios ampliamente conocidos como puede ser AWS o Amazon Web Services.

Con una rápida búsqueda por internet, se pueden encontrar los precios actuales de los servicios mencionados. En general, los precios pueden encontrarse alrededor de los 10 y los 30 euros al mes. Esto permite el acceso a una máquina virtual sencilla y sin mucha capacidad, sin embargo, también elimina la necesidad de mantenimiento por parte del desarrollador. Dependiendo del videojuego en cuestión, un desarrollador puede encontrarse con el mismo problema que tendría si tuviese que construir físicamente sus propios servidores, ya que es muy posible que los recursos de la máquina no cubran todas las necesidades del proyecto. Para remediar esta carencia, estos servicios comenzaron a ofrecer máquinas adaptativas por las que se factura dependiendo del uso de la misma, basado en diferentes factores. Gracias a esto se puede estar seguro de que los servidores serán suficientemente potentes pero también serán más costosos.

A lo largo de esta memoria se analizará la investigación realizada para encontrar una nueva solución que abarque ambos problemas. La idea es proporcionar facilidades al desarrollo de la aplicación de servidor, mientras que se optimiza el tráfico que pasa por el mismo, reduciendo así su coste. Para ello se creará una plantilla de servidor que utilizará varias tecnologías de las que se hablará más adelante. Gran parte de ellas irán enfocadas a permitir la comunicación en tiempo real entre clientes, permitiendo que el servidor quede a un lado durante la partida.

El proyecto contará con distintos componentes que necesitarán varias tecnologías y herramientas. Cada una de las partes del desarrollo en cuestión, requerirá de una investigación previa y un análisis comparativo de los resultados para encontrar la tecnología más adecuada. Tras la selección de herramientas, estas serán probadas individualmente para asegurarse de que su uso es viable para el proyecto final y, tras su implementación, se analizará su eficiencia y su funcionamiento.

Al finalizar la investigación, las conclusiones de la misma fueron la base del desarrollo de una herramienta que se encuentra en el repositorio de GitHub del proyecto (<https://github.com/JaviRegui/WebRTCgameServer>). Su propósito es ser clonado como una base ampliamente adaptable para que otros proyectos comiencen con funcionalidades básicas, como la asignación de salas de juego y la comunicación entre clientes mediante una nueva arquitectura que utiliza WebRTC, una tecnología que ayudará a reducir el coste de la infraestructura de los servidores.

La siguiente memoria comenzará marcando los objetivos principales de la investigación. A continuación, se establecerá el marco de la misma, exponiendo las características de las tecnologías existentes y su utilidad para el proyecto. Además, también se describirá la metodología de investigación y el proceso de desarrollo de la herramienta. Finalmente, se realizará un análisis de las conclusiones y resultados finales y se hablará del futuro del proyecto.

2. Objetivos generales

Los propósitos de la presente investigación son los siguientes:

- Investigar y analizar los diferentes aspectos del funcionamiento de los servidores de videojuegos y las tecnologías disponibles en dicho campo.
- Determinar, de manera experimental, un método de comunicación que presente ventajas en comparación a los métodos de desarrollo actuales en lo que se refiere a videojuegos multijugador.
- Utilizar los conocimientos adquiridos para desarrollar una herramienta gratuita de la que otros proyectos puedan beneficiarse y en la que se genere un back end con sistemas comúnmente encontrados en servidores de videojuegos, además de que utilice tecnologías web para mejorar la eficiencia de la comunicación durante la partida.

3. El estado de la cuestión

El tema a tratar en la siguiente investigación es el funcionamiento del proceso de desarrollo del back end de videojuegos multijugador, las herramientas usadas y su eficiencia. Las ideas que se extraigan serán la base del desarrollo de la herramienta propuesta. Es por esto que se debe comprender en profundidad el funcionamiento de las tecnologías actuales y las implicaciones del uso de cada una de ellas.

En primer lugar, es importante entender que un videojuego no es más que un tipo de aplicación. Puede que este ejecute un bucle de juego que requiera cierta velocidad, sin embargo, esto es solamente un aspecto a tener en cuenta como diferencia ya que, por lo demás, los principios en los que se basa cualquier aplicación que conecte varios dispositivos pueden extrapolarse a un videojuego multijugador, es decir, las arquitecturas y herramientas utilizadas pueden ser las mismas o muy similares, dependiendo de las necesidades del juego en cuestión. Con esta premisa, pueden empezar a analizarse las diferentes estructuras que permiten la comunicación en una aplicación. De esta manera, también se pueden diferenciar sus partes, analizarlas y entender la utilidad.

Arquitecturas de comunicación

Un aspecto de gran importancia, antes de indagar en las distintas tecnologías y protocolos usados en el desarrollo de aplicaciones multijugador, son las distintas arquitecturas de comunicación. Estas son los distintos diseños u organizaciones que pueden tomar las estructuras que permiten conectar varios dispositivos a través de una red, es decir, son una especie de mapa que representa los distintos componentes de la red, sus funciones y cómo se comunican entre ellos.

Dentro de una arquitectura de comunicación, los distintos dispositivos son denominados nodos. Además, simplificando la estructura de dicha arquitectura, los nodos se suelen clasificar en dos tipos principales: los clientes y los servidores. Esta clasificación separa a los miembros de la red según su función y según el tipo de nodo con el que pueden comunicarse dentro de la red. En cuanto a la representación gráfica de dichas estructuras, Arcenio y Gómez (2008) afirmaron:

Las arquitecturas de comunicación pueden ser elegidas a partir de diferentes modelos las cuales pueden ser organizadas de acuerdo a su grado de despliegue. En un grafo de comunicación los nodos representan los procesos ejecutados en computadoras remotas y las aristas indican que los nodos pueden intercambiar mensajes. (p.10)

Por otro lado, Arcenio y Gómez (2008) también aportan definiciones para ambos tipos de nodo en el ámbito de los videojuegos multijugador:

El servidor es una sesión que inicia y finaliza juegos, y acepta nuevas conexiones de clientes. Este programa es responsable de recibir todas las entradas de los clientes, calcular todo el estado del juego y mantener a los jugadores actualizados. El servidor realiza además los cálculos físicos que gobiernan el movimiento de las entidades y ejecuta la Inteligencia Artificial de los personajes. (p. 16)

El cliente es el programa con el cual el jugador interactúa, ejecuta los gráficos y los sonidos además de manipular la entrada de los dispositivos como teclado, mouse y joystick. Por lo tanto el cliente es responsable de enviar la entrada del jugador al servidor y de esperar por la respuesta de éste último para actualizar la vista del juego. (p.17)

En el mundo de los videojuegos multijugador, las arquitecturas de comunicación utilizadas han evolucionado a lo largo de los años, avanzando de la mano de crecientes necesidades de los proyectos.

Originalmente, los videojuegos multijugador ocurrían fuera de una red, de hecho, si se hablase en los términos expuestos anteriormente, se podría decir que utilizaban una arquitectura con un solo nodo de cliente. Esta es la arquitectura más sencilla. En ella, la única pantalla disponible para la aplicación se dividía en el número de partes que se necesitasen para mostrar la perspectiva de los jugadores involucrados. Poco a poco, otra arquitectura de comunicación comenzó a ser más utilizada: la arquitectura punto a punto, también conocida como "Peer to Peer" o "P2P". El punto a punto permitía conectar dos jugadores

directamente, y gracias a esto, se convirtió en un estándar para los juegos multijugador y especialmente los basados en turnos. También es posible utilizar el punto a punto para conectar a varios jugadores, pero la sincronización de los estados de juego adquiere mayor complicación, por lo que se considera un sistema reservado para situaciones específicas. Ambas arquitecturas siguen siendo utilizadas en la actualidad pero con sus clásicas limitaciones. Grandes ejemplos pueden ser los Call of Duty, juegos en los que se sigue utilizando la división de pantalla, por ejemplo, para el modo zombies. Además, fuera del mundo de los videojuegos, el punto a punto sigue siendo ampliamente utilizado en los programas de descarga de archivos Torrent, entre otros.

Posteriormente comenzó a utilizarse lo que se conoce como modelo cliente servidor, explicado por Arcenio y Gómez (2008):

El modelo cliente servidor (CS) es la arquitectura más usada actualmente en el desarrollo de videojuegos. En esa arquitectura hay un nodo especial (el servidor) al cual se conectan los demás nodos (los clientes), es por ello que en general la arquitectura sea centralizada. Son varias las razones por las cuales la arquitectura cliente servidor centralizada es tan popular. En primer lugar, CS es simple de implementar en relación con las demás arquitecturas, pues sólo se tiene en cuenta un tipo de conexión: la existente entre el cliente y el servidor. (p. 14)

La gran mayoría de arquitecturas restantes son derivados del modelo cliente servidor. Estas variantes pretenden arreglar problemas como la escalabilidad o la seguridad de los paquetes enviados mediante la estructura cliente servidor. Las dos maneras de escalar un sistema de servidores son el escalado vertical y el horizontal. El escalado vertical significa usar el mismo número de máquinas, pero modificar sus características para mejorar su rendimiento y su capacidad para realizar sus funciones, es decir, añadir una CPU mejor, más RAM, etc. Por otra parte, el escalado horizontal implica añadir más máquinas para que trabajen de manera conjunta y realicen las funciones del servidor. Este tipo de escalado tiene sus ventajas, por ejemplo, se aumenta considerablemente el tráfico que el sistema puede manejar y, en caso de que un servidor falle, otro puede cubrir sus funciones. Sin embargo, esto cambia la arquitectura de la red creando variantes del modelo cliente servidor, como la red o el pool de servidores. Un inconveniente de los mismos, es que también se debe implementar una lógica llamada balanceado de carga, que reparte el tráfico y las tareas entre los diferentes servidores incluso en caso de fallo.

En los últimos años, nuevos servicios relacionados con el back end, que serán explorados más adelante, han permitido crear una nueva estructura diferente a las ya mencionadas. Estas tecnologías se denominan “serverless” y representan un paradigma diferente a la hora de desarrollar aplicaciones en red. También se podría considerar una variante distribuida del modelo cliente servidor, ya que funciona con microservicios que se ejecutan en diferentes servidores cuando se necesitan. Según Espinosa (2018):

Una de las últimas técnicas desarrolladas es la posibilidad de aprovisionar funcionalidad sobre demanda sin la necesidad de aprovisionar servidores; a esto se le ha llamado en los últimos años tecnología Serverless y se ha vuelto más popular cada año desde la aparición de AWS Lambda en 2014. (p. 19)

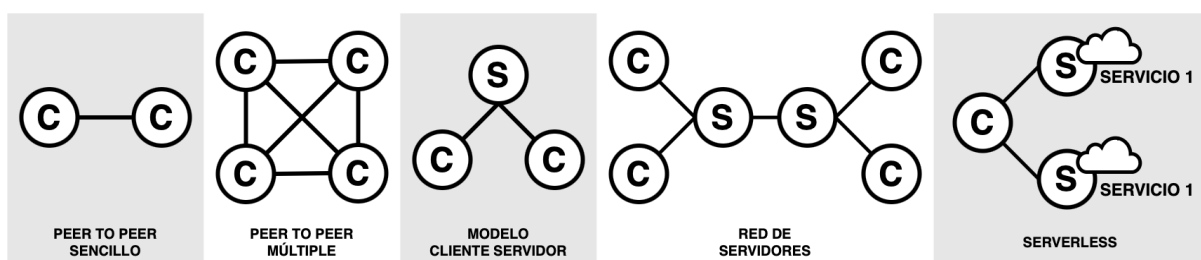


Figura 1: Arquitecturas de comunicación

Transporte: TCP y UDP

Un concepto importante a la hora de hablar de redes es el modelo OSI o modelo de interconexión de sistemas abiertos. Este fue creado en los años 80 y representa los procesos desarrollados durante la comunicación de dos dispositivos en una red. Para ello, estos procesos son divididos en siete capas. Cada una de las capas tiene una función y contiene los protocolos que pueden desempeñar dicha función. Un protocolo es un conjunto de normas que establecen el formato que seguirán los mensajes enviados entre dos clientes para que ambos puedan entenderlos. Las capas del modelo OSI son las siguientes:

- Física: convierte los datos en una señal y los transmite mediante un medio físico, como un cable de cobre o de fibra óptica, que conecta dispositivos.
- Enlace de datos: permite a los dispositivos conectados mediante la capa física acceder a la capa de red y se comunica con dichos dispositivos, por ejemplo, mediante sus direcciones MAC (única para cada dispositivo).
- Red: controla una red, evitando su congestión y diseñando las rutas para comunicar a sus miembros. También divide los mensajes en paquetes que deberá reconstruir al otro lado de la comunicación.
- Transporte: divide los paquetes y los transmite de extremo a extremo siguiendo normas como el orden de entrega de datos. Además se asegura de la fiabilidad de la transmisión y del flujo correcto de los datos.
- Sesión: abre y cierra las sesiones de comunicación entre dispositivos. En ocasiones también se encarga de la autenticación.
- Presentación: se encarga del cifrado, compresión y traducción de los datos para que las aplicaciones en ambos extremos de la comunicación puedan comprenderlos.
- Aplicación: permite al usuario final interactuar con la red para acceder a los recursos que buscan. Para ello, se comunica directamente con las aplicaciones.

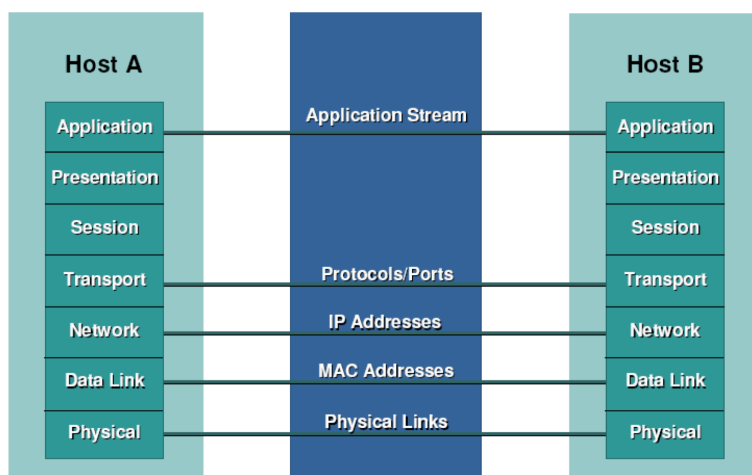


Figura 2: Modelo OSI

Como ya se ha mencionado, el modelo OSI es una simple representación de una realidad: la comunicación entre dispositivos. Sin embargo, existen otros modelos para representar esa misma realidad o, al menos, parte de la misma. Cuando la comunicación se realiza mediante Internet se puede utilizar otro modelo más simplificado llamado TCP/IP, nombre que adquiere por los protocolos más importantes que lo componen. El modelo TCP/IP, a diferencia del OSI, consta de cuatro capas:

- Interfaz de red: envía datos utilizando un medio físico pero no establece conexiones.
- Internet: al igual que la capa de red del modelo OSI, comunica dos dispositivos y crea la ruta que seguirá la información de los mensajes entre ellos.
- Transporte: en este caso, la capa comparte funciones e incluso nombre con su equivalencia del modelo OSI.
- Aplicación: cumple con las funciones de las tres últimas capas del modelo OSI, es decir, recibe los datos, los procesa y los muestra al usuario a través de las aplicaciones.

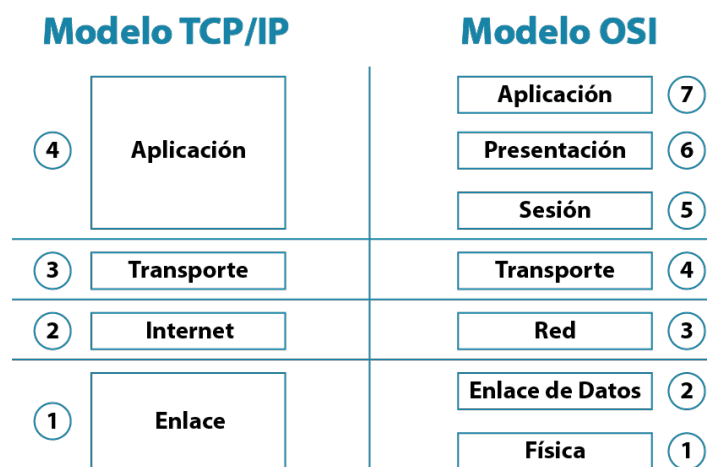


Figura 3: Modelo TCP/IP

Debido al tema de esta investigación, la capa de transporte es de especial interés. Esto se debe a que en dicha capa se encuentran dos protocolos con funcionamientos diferentes: TCP y UDP. La cuestión es que TCP, por motivos que serán estudiados más adelante, es utilizado para lo que se conoce coloquialmente como internet, es decir, es la base del protocolo HTTP y HTTPS, localizados en la capa de aplicación. Por otro lado, UDP es utilizado en casos en los que se necesita comunicación más rápida, por ejemplo, servidores de videojuegos.

El protocolo de datagramas de usuario, o “UDP”, se considera un protocolo de transporte no orientado a la conexión, debido a que los datos se envían sin necesidad de una conexión previa entre las partes y son transmitidos de manera desordenada. Además, UDP no tiene ningún mecanismo para asegurarse de que dichos archivos o datagramas (paquetes independientes de datos que no dependen de datagramas anteriores para poder ser transmitidos) han sido recibidos al otro lado. Esto representa una desventaja, ya que es relativamente poco fiable, sin embargo, también tiene la ventaja de producir una comunicación especialmente rápida, por lo que es el protocolo estándar a la hora de crear conexiones en tiempo real para

enviar video, audio o, especialmente, datos de juego. Un ejemplo de pérdida de datos que cualquiera puede conocer, son las pequeñas pausas en una videollamada por las que se pueden perder un par de frases de video o parte del audio.

Por otra parte, el “TCP”, o Protocolo de Control de Transmisión, se considera orientado a la conexión, ya que cuando dos dispositivos quieren comunicarse, crean una conexión, envían los datos en segmentos de manera ordenada, si alguno se pierde lo reenvía y, cuando se está seguro de que toda la información ha sido recibida correctamente, se cierra la conexión. Esto tiene ventajas y desventajas. Por ejemplo, este tipo de comunicación resulta mucho más fiable que la establecida mediante UDP, ya que se puede tener la seguridad de que toda la información enviada será recibida, sin embargo, todas las comprobaciones necesarias hacen que el proceso resulte más lento.

Ambos protocolos pueden ser utilizados en conjunto al protocolo HTTP, sin embargo, lo más común es el uso de TCP. En el caso de los videojuegos, la velocidad de UDP lo convierte en el protocolo más extendido, sin embargo, cuando se desarrollan juegos basados en tecnologías web, es necesario encontrar una implementación eficiente de comunicación en tiempo real entre clientes.

Comunicación en tiempo real: Web Sockets y WebRTC

Existen varias maneras de crear comunicación en tiempo real entre dispositivos. La idea general es crear una conexión entre las partes que se mantendrá abierta de manera constante. De esta forma, pueden evitarse varios pasos de un proceso de comunicación como el de peticiones y respuestas HTTP (REST). Por ejemplo, no todos los mensajes necesitan respuesta. También se puede reducir el tamaño de los mensajes enviados ya que las cabeceras con los datos de las partes dejan de ser necesarias.

Una de las opciones más conocidas son los Web Sockets. Aún cuando este protocolo permite usar tanto TCP como UDP, el uso de UDP requiere una configuración adicional. De todas formas, al crear una conexión en tiempo real se gana velocidad incluso al usarse TCP. Gracias a esto, el uso de Web Sockets aporta lo mejor de ambos mundos: la velocidad de una conexión en tiempo real, junto a la fiabilidad de un protocolo orientado a la conexión. De hecho, las diferencias pueden llegar a ser significativas según lo expuesto por Luecke (2018):

On average a single HTTP request took about 107ms and a Socket.io request 83ms. For a larger number of parallel requests things started to look quite different. 50 requests via Socket.io took ~180ms while completing the same number of HTTP requests took around 5 seconds. Overall HTTP allowed to complete about 10 requests per second while Socket.io could handle almost 4000 requests in the same time. (párr. 8)

En este fragmento, Luecke explica los resultados de sus experimentos sobre la velocidad de los Web Sockets. Estos resultados indican que una petición que utilice esta tecnología resulta un 23% más rápida que una petición HTTP. Sin embargo, cuando el número de peticiones simultáneas es mayor, la diferencia se dispara. En sus experimentos, 50 peticiones producían una diferencia del 400%, es decir, HTTP completaba 10 peticiones por segundo mientras que Web Sockets llegaba a las 4000. En ese mismo artículo, se puede observar un gráfico que muestra el peso de los mensajes enviados durante el experimento, otro aspecto en el que destaca el protocolo Web Sockets:

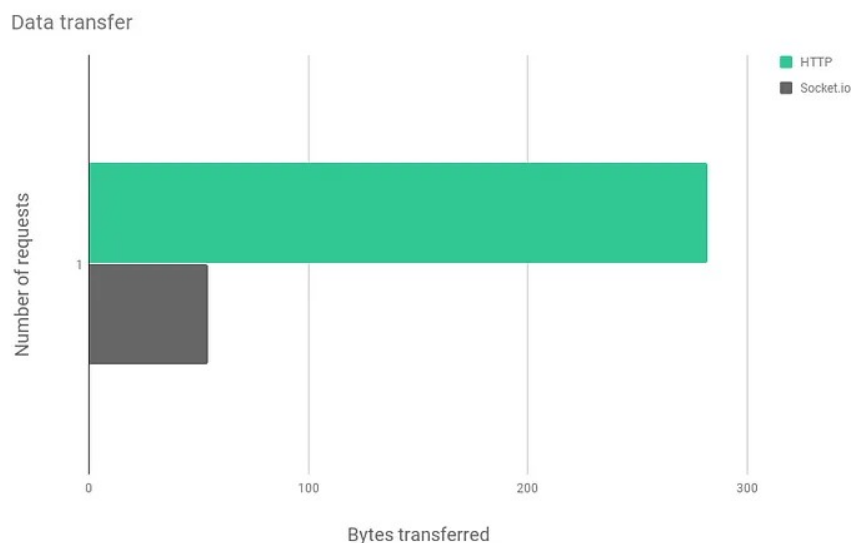


Figura 4: Transferencia de datos HTTP vs. Socket.io

El protocolo Web Socket ya es un estándar en gran parte de los navegadores, además, fue normalizado por el Grupo de Trabajo de Ingeniería de Internet (IETF) en el año 2011 tras la publicación del RFC 6455, un documento que formaliza las normas que sigue dicho protocolo y el funcionamiento del mismo.

Los sockets utilizan una conexión full-duplex, esto significa que el canal de comunicación es bidireccional, es decir, ambas partes pueden enviar y recibir mensajes al mismo tiempo. Esta conexión se consigue mediante lo denominado “handshake”, realizado mediante una negociación formada por una serie de solicitudes HTTP que contienen la información necesaria de los miembros que quieren abrir el canal de comunicación.

En los experimentos de Luecke previamente mencionados, la tecnología utilizada es Socket.io. Esta es una implementación en JavaScript del protocolo Web Socket, que permite utilizarlo en servidores de Node.js mediante un módulo accesible desde npm, un administrador de paquetes que ayuda a añadir funcionalidades a los servidores de node. Socket.io también simplifica el desarrollo del servidor Web Socket y proporciona el concepto de salas de sockets conjuntos, además de broadcasting dentro de las mismas, entre otras funciones. Además, se debe recalcar que Socket.io es una implementación del protocolo Web Socket, no el propio protocolo, por esto el manual del mismo indica que los paquetes enviados contienen más información de la necesaria en un socket tradicional. Si los experimentos de Luecke se hubiesen realizado con el protocolo Web Socket y sin Socket.io los resultados serían aún más favorables.

Although Socket.IO indeed uses WebSocket for transport when possible, it adds additional metadata to each packet. That is why a WebSocket client will not be able to successfully connect to a Socket.IO server, and a Socket.IO client will not be able to connect to a plain WebSocket server either. (Socket.IO, 2023, párr. 2)

Una de las limitaciones del protocolo Web Socket es que produce exclusivamente conexiones entre clientes y un servidor dedicado a las mismas. Sin embargo, actualmente existe un método para establecer conexiones en tiempo real que no comparte esta limitación, es decir, una tecnología llamada WebRTC que permite crear conexiones directamente entre clientes y prescindiendo del servidor. WebRTC es un proyecto

de código abierto basado en una serie de APIs que permiten compartir voz, video o datos mediante canales en tiempo real, muy parecidos a Web Sockets.

Para los propósitos de esta investigación, el punto más interesante de WebRTC es la posibilidad de utilizar un “data channel” para enviar texto en tiempo real entre dos navegadores. Esto puede ser útil en el mundo del videojuego como afirma Martín (2016):

La API RTCDataChannel representa un canal bidireccional de intercambio de datos entre Peers. Esta herramienta aumenta en gran medida la utilidad de WebRTC, pudiendo extenderse a aplicaciones como videojuegos online o transferencia de archivos. (p. 21)

La conexión WebRTC se consigue de manera parecida a la del protocolo Web Socket en el sentido de que también debe ocurrir una negociación para abrir el canal de comunicación. Para que dicha negociación ocurra, se necesita algún método de señalización. Normalmente, esto se realiza mediante un servidor que puede utilizar otras tecnologías como Web Socket y, una vez este cumple su función, deja de funcionar hasta que vuelva a ser necesario. En el caso de utilizar Web Sockets para la señalización, cabe destacar que, en el momento de reposo del socket, este sigue consumiendo recursos. Sin embargo, este consumo es despreciable en el cómputo general del funcionamiento de una aplicación. Especialmente si es comparado con el gasto que supondrían otros sistemas como HTTP.

Lo primero que debe ocurrir para crear una conexión WebRTC es que se generen en los clientes instancias de RTCPeerConnection, objetos que contienen lo necesario para establecer la conexión. Como el nombre del objeto indica, WebRTC es una implementación más avanzada de la arquitectura Peer to Peer. A continuación, uno de los clientes debe crear una oferta, un archivo de tipo SPD (Session Description Protocol) que contendrá información sobre él mismo y sobre las diferentes rutas existentes para conectarse a él de manera directa. A esas rutas se les denomina candidatos ICE y, en caso de que la red cambie y exista una nueva ruta, esta será añadida a la información de la oferta. Una vez creada la oferta, esta será enviada al otro cliente por medio del servidor de señalización y, al recibirla, el otro cliente creará una respuesta con sus datos. Cuando el servidor de señalización envíe la respuesta al cliente original, su trabajo habrá terminado y el primer cliente establecerá el canal de comunicación.

Un problema común en las conexiones WebRTC es que, para conectar dos clientes de forma directa, estos deben ser accesibles entre ellos. Muchas veces, un cliente se encuentra detrás de la seguridad de su propia red, por lo que se crea confusión con el NAT y las IPs privadas, entre otros. Una posible solución sería situar ambos dispositivos en la misma red, de esta manera todos estos conceptos serían irrelevantes, sin embargo, en un escenario real raramente ocurre que todos los jugadores se encuentren en el mismo lugar y en la misma red. Para solucionarlo, existen unos tipos de servidor disponibles públicamente llamados STUN y TURN que ayudan a los dispositivos a encontrarse entre ellos. Martín (2016) los define de la siguiente manera:

El protocolo de red STUN (Session Traversal Utilities for NAT) envía paquetes de test previos al establecimiento de la sesión multimedia para permitir al navegador saber si esta detrás de un NAT y obtener su dirección IP desde una perspectiva pública. (p. 27)

El protocolo TURN (Traversal Using Relays around NAT) es una extensión de STUN. Se utiliza para retransmitir información entre los Peers si RTCPeerConnection falla en el establecimiento de la comunicación entre estos.

En algunos casos podemos encontrar firewalls muy restrictivos que no permitan el intercambio de candidatos ICE, o incluso bloqueen el tráfico multimedia (UDP) entre los Peers. Una solución en este

caso es incluir un servidor TURN permitido por el firewall que retransmita la información multimedia.

Todo servidor TURN puede ser utilizado como un servidor STUN con la funcionalidad añadida de retransmisión de tráfico multimedia entre los Peers. (p. 28)

WebRTC es un protocolo seguro y abalado por gigantes de la industria tecnológica. Sin embargo, el proceso de señalización, el cual depende del método implementado en cada caso, puede ser menos seguro. Dentro de la oferta se puede encontrar información sensible del cliente. Por este motivo es interesante investigar métodos de comunicación seguros.

Seguridad: El modelo Tor

A la hora de ocultar mensajes es necesario algún tipo de proceso de encriptado. Existen cientos de herramientas que realizan procesos externos a las aplicaciones en las que son utilizadas para asistir en la tarea de encriptar y desencriptar datos fuera de la base de código del proyecto.

Sin embargo, hay otro aspecto más complicado en lo que se refiere a la seguridad dentro de una red y los riesgos que corren los datos que se transfieren por ella. Un gran enfoque para afrontar dichos riesgos es el que toma el navegador Tor. Tor es un navegador usado para ocultarse en Internet. Gracias a esto, es el navegador estándar para acceder a lo que se conoce como la "deep web". Su funcionamiento es relativamente sencillo: cuando un cliente hace una petición, esta petición no va directamente al destinatario. Antes pasa por otros clientes que ocultarán los datos del solicitante y enviarán los recursos solicitados de vuelta una vez los recuperen del servidor en el que se encuentran. Además, se debe tener en cuenta que todas las comunicaciones entre los clientes se encuentran cifradas. A esto se le denomina Tor Net y explicado por el propio manual (s.f.):

La red Tor es una red de túneles virtuales que permite mejorar la privacidad y seguridad en Internet. Tor funciona enviando el tráfico a través de tres servidores aleatorios (también conocidos como repetidores) en la red Tor. El último repetidor en el circuito (el "repetidor de salida") envía el tráfico hacia el Internet público.

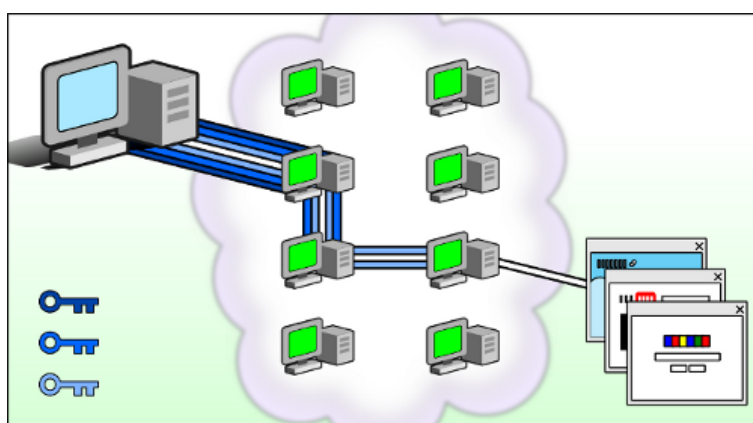


Figura 5: Diagrama Tor Net

Alquiler de servidores: Infraestructura como servicio

A lo largo de esta investigación se ha comprobado que, a la hora de crear un videojuego multijugador con una experiencia de usuario aceptable y tecnologías que simplifiquen el desarrollo del mismo, es necesario el uso de servidores y, debido a los objetivos de la investigación, se debe buscar una forma sencilla, barata y eficiente para crear a esta infraestructura.

Aún cuando cualquier ordenador puede actuar como servidor con una rápida instalación de Apache o Ngix, un servidor de videojuegos puede ser más complejo. Montar el hardware no siempre es tarea fácil, la configuración para abrirlo a internet mediante puertos puede tener riesgos de seguridad, un aspecto que hay que tener muy en cuenta a la hora de realizar el mantenimiento que constantemente requerirá. Además, todo este proceso, o incluso los componentes, pueden ser costosos a nivel económico. Por estos motivos y muchos más, montar servidores no suele ser la opción elegida, de hecho, a día de hoy incluso grandes empresas confían en el Cloud Computing y en la infraestructura que otros proporcionan para alojar sus aplicaciones. Otra ventaja de estos servicios es que, en caso de poseer servidores propios, su velocidad de conexión a la red puede no ser óptima. Además, los servidores se encontrarían en un punto geográfico específico y carecería de flexibilidad en ese sentido por lo que, conexiones desde lugares lejanos podrían carecer de una velocidad aceptable.

El concepto de Cloud Computing ha crecido mucho en los últimos años y engloba muchos otros servicios de entre los cuales, en esta memoria, se analizará el alquiler de los servidores en si. A esto se le llama "Infrastructure as a Service", "IaaS" o "Hardware as a Service" o "HaaS".

Este servicio es generalmente proporcionado por grandes empresas en el mundo de la informática y funciona mediante máquinas virtuales. Esto quiere decir que, al alquilar un servidor, se crea una instancia de una máquina virtual, una división en uno de los miles de servidores físicos que tienen dichas empresas distribuidos por el mundo, que actúa como un ordenador independiente dentro del mismo. En esta máquina virtual, el desarrollador puede hacer lo que quiera, siempre que cumpla con las condiciones de la empresa proveedora del servicio. La compartimentación creada mediante el uso de máquinas virtuales o el mantenimiento automático realizado en el servidor, son motivos por los que estos servicios simplifican las tareas de los desarrolladores. Además, también se debe tener en cuenta que empresas como Amazon gozan de una gran infraestructura, como puede ser una red de fibra óptica privada, que les permite decidir quién entra en su red. De esta manera, pueden mejorar la seguridad de sus servidores y las aplicaciones alojadas en ellos, otro aspecto tremendamente importante para estos servicios y que resta una gran carga de trabajo a los desarrolladores.

Los grandes proveedores en este ámbito son las empresas que comenzaron a ofrecer servicios de Cloud Computing a finales de los 2000 y principios de los 2010. Cada una de ellas tiene un método para calcular sus precios. Estos dependen del servicio que se busque, ya que la oferta ha crecido a lo largo de los años incluyendo bases de datos, autenticación, autoescalado, etc. En el caso de los propios servidores y el alojamiento o "hosting" de una aplicación de servidor, los distintos proveedores suelen tener dos opciones principales, una de alquiler de máquinas virtuales y otra basada en microservicios serverless. Los principales proveedores de estos servicios son:

- Amazon Web Services (AWS): lanzado en 2006, fueron los primeros de la industria. Ya se considera un estándar, incluso con su grandísima curva de aprendizaje. Su servicio de alquiler de servidores se llama Elastic Compute Cloud o "EC2", mientras que su servicio de funciones serverless se denomina Lambda. Los servidores son gratuitos el primer año, esto se debe a que ofrece 750 horas al mes gratis durante los primeros 12 meses. Tras el primer año, un servidor Linux básico en EC2 tiene precios relativamente bajos dependiendo del tiempo de uso en horas. Estos proporcionan 2 procesadores virtuales o vCPU y 0,5 GB

de memoria. Este servicio también permite alquilar servidores mucho más potentes que pueden costar hasta más de 100 dólares por hora aún con las distintas ofertas y planes de ahorro de los que se dispone. Por otro lado, Lambda puede ser gratuito para siempre dependiendo del tráfico que se espere. En este servicio, el primer millón de solicitudes al mes son gratuitas y, a partir de ahí, cada millón de solicitudes tiene un coste.

- Google Cloud Platform: se creó en 2008 y sigue un modelo muy similar al de Amazon. Su Compute Engine permite alquilar servidores sencillos por hora con 1 procesador virtual y 3,75 GB de memoria. Al igual que AWS, también tiene oferta de servidores más potentes. Además, según pasa el tiempo, se ofrecen descuentos por continuidad de uso sin necesidad de anticipar el tiempo que se necesitarán los servidores. Su servicio de funciones serverless, conocido como Cloud Functions, regala los 2 primeros millones de solicitudes al mes y el resto de solicitudes se pagan por millón al igual que en Amazon. Sin embargo, cabe destacar que los precios de google tienden a ser más bajos.
- Microsoft Azure: comenzó a funcionar en 2010 y es la alternativa de Microsoft en cuanto a Cloud Computing. Una de sus máquinas virtuales con 1 vCPU y 0,5 GB de memoria, a diferencia de los servicios anteriores, se alquila por mes tras el primer año, que es gratis. La oferta serverless de Microsoft se llama Azure Functions y ofrece un millón de solicitudes gratuitas al mes tras lo que se paga por millón de solicitudes.

Esto debería dar una idea del funcionamiento de los servicios mencionados y sus sistemas de cobro, basados en el tiempo de uso y el uso de recursos en ese tiempo. Sin embargo, los datos cambian con el tiempo y los precios no son siempre los mismos. Es por esto que se ha evitado citar precios exactos, de todas maneras, para observar la relación entre servicios, existe un estudio por RightScale (2016) que mostraba la situación en ese momento.

AWS vs. Azure vs. Google On-Demand Prices

Resource Type (us-east, Linux)	AWS Instance	Azure Instance	Google Instance	AWS OD Hourly	Azure OD Hourly	Google OD Hourly	AWS /GB RAM	Azure /GB RAM	Google /GB RAM
Standard 2 vCPU w SSD	m3.large	D2 v2	n1-standard-2	\$0.133	\$0.114	\$0.212	\$0.017	\$0.016	\$0.028
Highmem 2 vCPU w SSD	r3.large	D11 v2	n1-highmem-2	\$0.166	\$0.149	\$0.238	\$0.011	\$0.011	\$0.018
Highcpu 2 vCPU w SSD	c3.large	F2	n1-highcpu-2	\$0.105	\$0.099	\$0.188	\$0.028	\$0.025	\$0.104
Standard 2 vCPU no SSD	m4.large	D2 v2	n1-standard-2	\$0.108	\$0.114	\$0.100	\$0.014	\$0.016	\$0.013
Highmem 2 vCPU no SSD	r4.large	D11 v2	n1-highmem-2	\$0.133	\$0.149	\$0.126	\$0.009	\$0.011	\$0.010
Highcpu 2 vCPU no SSD	c4.large	F2	n1-highcpu-2	\$0.105	\$0.099	\$0.076	\$0.027	\$0.025	\$0.042

As of Dec 2, 2016

Source: RightScale

Figura 6: Precios Cloud Computing

Soluciones disponibles para el desarrollo de servidores

Con toda la información previa, es evidente que desarrollar el back end de un videojuego multijugador no es tarea fácil. Es por esto que sería lógico asumir que existen herramientas online que simplifiquen el proceso, sin embargo, estas soluciones suelen estar incompletas o requieren suscripción. No existe una herramienta gratuita que enfrente las múltiples dificultades que presenta la creación de los servidores de un juego web.

Entre las herramientas que cumplen esta función destaca Colyseus Multiplayer Framework o “Colyseus”, que según su página web (2018) “Colyseus is a framework for writing your own authoritative multiplayer game servers using JavaScript/TypeScript and Node.js, and easily integrating it with your favorite game engine” (párr. 1). Este framework permite crear aplicaciones de servidor flexibles para cada proyecto. También es de código abierto y puede ser alojada en cualquier servidor, sin embargo, también se ofrece la opción de pagar por un hosting incluido. Además, existen otras opciones cuyo objetivo es dar herramientas sencillas para el desarrollo de la aplicación de servidor, sin embargo, las alternativas encontradas son proyectos sin terminar. Una de ellas se encuentra en npm bajo el nombre de “game-room-server” y fue abandonada hace 4 años. Por otro lado, existe un repositorio de GitHub en versión alfa llamado “node-gameroom” cuya última contribución se realizó en el año 2010 por el usuario “aheckmann”.

Todas estas herramientas tratan de solucionar alguno de los desafíos que presenta la creación de un servidor de videojuegos, sin embargo, no son soluciones completas. Tienden a centrarse en simplificar el proceso de desarrollo creando plantillas de mecánicas necesarias en la mayoría de los casos, como un sistema de salas para el que se suele utilizar el protocolo Web Sockets y Node.js. Es un enfoque correcto si el único objetivo es agilizar el desarrollo, sin embargo, también existe el problema de la infraestructura, su complejidad y su coste. Una posible mejora de las herramientas existentes podría ser la implementación de un sistema que utilice comunicación en tiempo real entre clientes, como WebRTC, para crear una arquitectura que permita que el tráfico de la partida no pase por el servidor central, manteniendo las ventajas en cuanto a simplicidad y adaptabilidad al proyecto, pero añadiendo la ventaja del ahorro en IaaS. En este aspecto, existen tutoriales y vídeos en YouTube dirigidos al desarrollo de este tipo de comunicación, aunque tampoco abarcan todas las cuestiones mencionadas y no profundizan en el problema a tratar.

4. Metodología

Tras la investigación, el objetivo es desarrollar la herramienta descrita a continuación. Para ello, se debe analizar la idea del proyecto para encontrar sus componentes principales. Gracias a esto, se puede utilizar la información obtenida previamente para comparar las tecnologías que podrían lograr la funcionalidad deseada de dichos componentes y escoger la opción adecuada para el caso.

Una vez seleccionadas las herramientas, cada una de ellas debe ser investigada para comprender su proceso de implementación y sus características. Después, se implementa al proyecto y se crean pruebas para comprobar que cumplen con su cometido. También se utilizan las herramientas de análisis proporcionadas por el servicio de hosting para estudiar su eficiencia a nivel de velocidad y uso de recursos.

Cuando uno de los componentes de la aplicación ha sido probado con resultados adecuados, se puede pasar a la siguiente tarea y, al concluir el desarrollo de la herramienta en cuestión, se debe probar de manera global con diferentes dispositivos para asegurarse de su compatibilidad con los mismos antes de publicar el código en GitHub.

5. Desarrollo del proyecto

Roles de programación

Las tareas realizadas en cuanto a programación han sido variadas. Entre ellas, se encuentra parte de la implementación de la lógica de la partida del juego Save the King, es decir, el movimiento, las colisiones, la funcionalidad de la interfaz, etc. Sin embargo, la mayor aportación fue la construcción de los menús que controlan el sistema de salas y el back end de las mismas, ya que todo ello fue basado en la herramienta desarrollada gracias a esta investigación. A continuación se analizará dicha herramienta y su desarrollo.

Análisis

El back end generado para este proyecto debía cumplir con ciertas condiciones. Estas se corresponden con las barreras presentadas anteriormente en lo que se refiere al desarrollo de servidores de videojuegos, es decir, debe ser un sistema moldeable a otros proyectos, que simplifique el desarrollo del propio juego y que utilice comunicación en tiempo real entre clientes para aliviar la carga del servidor. Un aspecto importante para cumplir dichos requisitos era el uso de herramientas ampliamente extendidas. Es por esto que se optó por crear una plantilla de servidor de Node.js que otros desarrolladores puedan clonar como base de sus proyectos.

Node.js es tremendamente conocido y, al ser un entorno de ejecución de JavaScript, utiliza uno de los lenguajes de programación más populares en el mundo. Además, node permite instalar dependencias que amplían su funcionalidad. Las dependencias instaladas en este proyecto proporcionaron herramientas imprescindibles para el desarrollo del mismo. Express se encarga de la distribución de datos a los clientes a la hora de entrar en la web del juego, enviando los recursos correctos una vez son solicitados. Esta es otra herramienta que, junto con node, es ampliamente conocida por los desarrolladores. Otra herramienta incluida en el proyecto fue Socket.io. Como se menciona previamente, Socket.io es una implementación del protocolo Web Socket, sin embargo, suele tener un rendimiento menor ya que envía paquetes más grandes con datos necesarios para su funcionamiento. Elegir esta librería por encima del protocolo Web Socket fue una decisión guiada por los propios requisitos del proyecto, es decir, un aspecto importante era buscar herramientas sencillas y, Socket.io, proporciona facilidades en el desarrollo comparado con el protocolo en el que se basa. Estas herramientas representan las tres capas del funcionamiento del proyecto: el servidor web que envía a los clientes al lugar adecuado, el servidor de señalización y, por último, la comunicación entre clientes para la que se implementó WebRTC. Gracias a los Data Channels de WebRTC fue posible crear un sistema en el que los clientes se comuniquen entre ellos. Esto cumple con la idea de reducir el tráfico procesado por el servidor central pero, además, permite crear un par de funciones predeterminadas que simplificarán aún más el trabajo de los desarrolladores. Para cumplir con el tercer requisito, que el sistema sea adaptable a cualquier proyecto, dichas funciones siempre son accesibles por el desarrollador para poder adaptar la arquitectura de comunicación, el procesamiento de los datos a la hora de enviarlos y recibirlos o, especialmente, los datos que se envían, por ejemplo, un JSON con el estado del jugador para poder juntarlo al estado de juego general.

El uso de ciertas tecnologías, como Express o WebRTC, hacen que las plataformas compatibles con este sistema se vean limitadas al navegador. Esto no representa un problema, ya que el objetivo de la herramienta eran los videojuegos web para navegador como Save the King. Sin embargo, WebRTC es una tecnología relativamente nueva que, aunque cada vez es soportada por más dispositivos y navegadores, todavía no es compatible de manera universal.

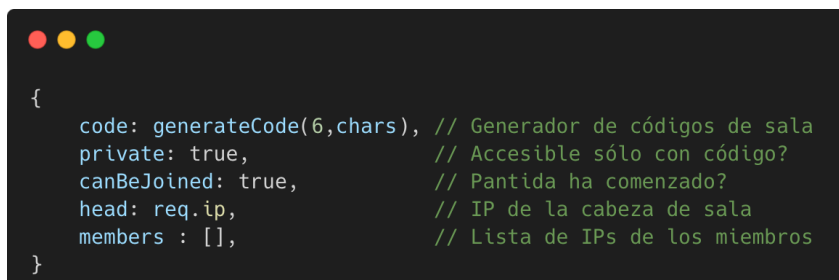
Relación de tareas

El proyecto desarrollado está formado por tres bloques en una especie de cascada, es decir, cada uno depende de los anteriores para funcionar. Cada bloque requirió de varias tareas y, a su vez, cada una de ellas consta de varios pasos e inconvenientes que se tuvieron que afrontar.

El primer bloque del proyecto fue la creación del propio servidor, su parte encargada de las solicitudes HTTP y el sistema de salas. Aún cuando se contaba con experiencia creando aplicaciones de servidor sencillas en python, se necesitó una investigación previa sobre Node.js, Express y tecnologías relacionadas. Tras esta investigación, se comenzó el desarrollo de una aplicación básica de Express en la que se configuró el motor de plantillas EJS para poder realizar lo que se conoce como “Server Side Rendering”. Esto significa que EJS se utiliza para crear plantillas con campos que el servidor rellenará con las variables adecuadas, una vez reciba una solicitud. De esta manera se puede adaptar el contenido recibido por cada cliente de una misma URL dependiendo de sus necesidades y modificar el front end de manera acorde.

A la hora de responder a las solicitudes HTTP que recibe el servidor, Express, permite usar un “router”, un archivo que, como su propio nombre indica, le da la ruta a seguir al cliente dependiendo de lo que haya pedido. En otras palabras, dependiendo de la URL a la que intente acceder el cliente, el servidor ejecuta unas funciones u otras y envía la respuesta adecuada. Un ejemplo claro son los botones encargados del sistema de salas que se explicará a continuación. Uno de estos botones dice “Buscar Sala” y, si el cliente lo presiona, hará una solicitud a la URL “http://abcde.com/findRoom”. Una vez el servidor recibe esa solicitud, el router ejecutará las funciones necesarias para encontrar una sala, añadirá al cliente a la misma y lo enviará a la URL “http://acbde.com/room/123ABC” para mostrar la sala en cuestión.

El último componente del primer bloque es un script llamado roomController.js. Este contiene una lista vacía que se irá llenando con objetos que describan las salas que se vayan creando.



```
{
  code: generateCode(6,chars), // Generador de códigos de sala
  private: true,               // Accesible sólo con código?
  canBeJoined: true,           // Partida ha comenzado?
  head: req.ip,                // IP de la cabeza de sala
  members : [],                // Lista de IPs de los miembros
}
```

Figura 7: Modelo de datos de la sala de juego

En la figura 6 se puede observar la estructura del objeto que describe la sala. Cabe destacar que cuando una sala es creada, llama al método generateCode para que cree un código de sala con el que será identificada. Además, el resto de propiedades explican las características de la sala: private indica si dicha sala será accesible al buscar una sala aleatoria o si solamente se podrá entrar mediante código. Por otra parte, canBeJoined se modificará cuando la partida comience y nadie más pueda entrar en la sala. Por último, existen dos variables que contendrán la información de los miembros, es decir, sus direcciones IP: la primera es members, una lista con todos los integrantes de la sala, y la segunda es head, una distinción entre uno de los miembros. El hecho de que cada sala tenga una cabeza tomará importancia más adelante.

Este script contiene las funciones relacionadas con las salas que serán invocadas por el router en caso de ser necesarias. Consta de dos métodos auxiliares que utilizan la información de la solicitud para averiguar qué

cliente la está realizando y buscar información sobre él. Estas funciones son: `getRoom`, encargada de encontrar la sala a la que pertenece el cliente, si es que pertenece a alguna, e `isMember`, encargada de asegurarse de que el cliente es parte de la sala a la que intenta acceder.

El resto de métodos son los invocados por los botones del front end. Estas son ejecutadas por el router y están encargadas de la administración de las salas, es decir, crearlas, permitir a usuarios encontrar una sala pública o enviar al usuario a la sala que busca mediante el código. Para ello, estas funciones deben realizar una serie de comprobaciones, como si hay sitio en la sala o si la partida ya ha comenzado.

- `createRoom`: crea una sala privada con el solicitante como cabeza.
- `findRoom`: recorre la lista de salas buscando una sala pública con espacio libre y preparada para recibir otro jugador.
- `joinRoom`: encuentra la sala según el código, realiza las comprobaciones pertinentes y envía al cliente a la sala.

Con esta primera etapa terminada, se cuenta con un front end y un back end que se integran para enviar a los clientes a las salas correctas. Esto significa que, si un cliente no está donde debe, se le reenviará al inicio mientras que los demás, si son miembros, podrán acceder a las URLs de las salas en las que se desempeñará el resto del proceso. A continuación, los resultados de esta primera etapa de desarrollo, debían ser probados. Para ello, se encontró un servicio de alojamiento muy económico y con una interfaz de usuario tremendamente sencilla: `Railway.app`. Las ventajas de este servicio, además de las ya mencionadas, fueron las herramientas de análisis proporcionadas que ayudaron a cuantificar los resultados del proceso de testeo. Esta plataforma también realiza deploys automáticos cuando el código del repositorio al que se conecta es modificado.

La segunda etapa del desarrollo se trataba de crear el servidor de señalización para las conexiones WebRTC, proceso que requirió una investigación sobre WebRTC y cómo implementarlo. Tras indagar en el funcionamiento de un servidor de señalización, se llegó a la conclusión de que se necesitaba comunicación en tiempo real entre el cliente y el servidor y, debido a las ventajas analizadas anteriormente, se decidió utilizar `Socket.io`. Una de las preocupaciones en este paso del desarrollo fue la seguridad. Esto se debe a que cualquier mensaje enviado o recibido mediante el socket puede observarse mediante las herramientas de desarrollo de los navegadores. Estas herramientas muestran detalles contenidos en el SDP de la oferta. De todas formas se optó por la sencillez de la herramienta aunque esta pueda ser mejorada en el futuro. El siguiente problema encontrado fue que, las tareas que debe realizar el socket, dependiendo del mensaje recibido, en muchos casos, debían ser ejecutados sobre las salas. Esto se solucionó haciendo referencia a la lista contenida en `roomController`, sin embargo, `Socket.io` diferencia a los clientes según el socket, no su IP. Debido a esta incompatibilidad, se creó un mapa, es decir, un conjunto de claves y valores para poder guardar las equivalencias entre IP y socket. De esta manera, esta nueva capa del programa podía comunicarse de manera correcta con la capa anterior y las salas que contiene.

Finalmente, se modificaron los archivos EJS de la sala para que los HTML finales de cada una ejecutasen un script que los conectase mediante sockets al servidor. Además, se probó el sistema para asegurarse de que los sockets funcionaban, detectaban cuando un cliente abandonaba la web para eliminarlo de la sala y crear un sistema de limpieza de salas vacías y si se podía mandar la información necesaria para crear conversaciones entre clientes. Es decir, un cliente debe poder comunicarse con el servidor, pero también debe poder mandar un destinatario junto al mensaje para que el servidor se lo reenvíe al cliente correcto. Este concepto es la base de un servidor de señalización y un paso imprescindible para implementar la tercera y última capa del proyecto, es decir, WebRTC.

Por último, se utilizó la base creada mediante sockets para implementar la negociación de WebRTC. En este punto toma importancia el concepto de cabeza de sala. La herramienta está pensada para ser modificada por el desarrollador que la necesite, sin embargo, la arquitectura de comunicación inicial puede ser uno de los aspectos que más trabajo lleve modificar ya que, dicha arquitectura, tiene sus raíces en la primera etapa del desarrollo y en el propio concepto de sala, que ya cuenta con una cabeza. Esta arquitectura convierte cada sala en una especie de instancia del modelo cliente servidor. La cabeza de la sala es el jugador con el que el resto de clientes deben establecer una conexión peer to peer. Esto significa que este encargado tendrá responsabilidades diferentes a las del resto de jugadores cuya lógica se debe implementar en el lado del cliente.

Esta arquitectura en la que existe una cabeza de sala, presenta otro reto. En un escenario de juego real, el cliente encargado, puede tener fallos de conexión o incluso puede abandonar la partida. La solución implementada se reduce a enviar a todos los miembros de la sala al menú principal. Sin embargo, sería interesante estudiar la opción de crear un sistema en el que, si fuese necesario, la cabeza de sala podría cambiar de un miembro a otro sin necesidad de pausar la ejecución del juego. Para ello, habría que diseñar un sistema dinámico de conexiones en tiempo real entre clientes o replantearse la totalidad de la arquitectura de comunicación implementada actualmente.

Cuando un cliente llega a una sala, el socket le preguntará si es la cabeza de la misma, de ser así, su trabajo habrá terminado hasta que otro cliente entre en la sala. Por el contrario, si el cliente correspondiente con el socket creado no es la cabeza, este enviará una petición de oferta al servidor mediante el socket. El servidor deberá utilizar el socket para localizar la IP del cliente y usarla para encontrar su sala y la cabeza de la misma. A continuación, el servidor enviará la petición de oferta al socket relacionado con la IP de la cabeza de sala, dando comienzo a una conversación entre ambas partes. Durante esta conversación, en la que el servidor todavía toma parte como intermediario, la cabeza abrirá un data channel y se preparará para establecer una conexión mediante WebRTC. También generará una oferta y se la mandará al cliente. El cliente seguirá el mismo proceso para generar una respuesta que, al ser recibida por la cabeza de sala, abrirá la comunicación entre ambos. Esto permitirá que el servidor abandone la mayoría de sus funciones.

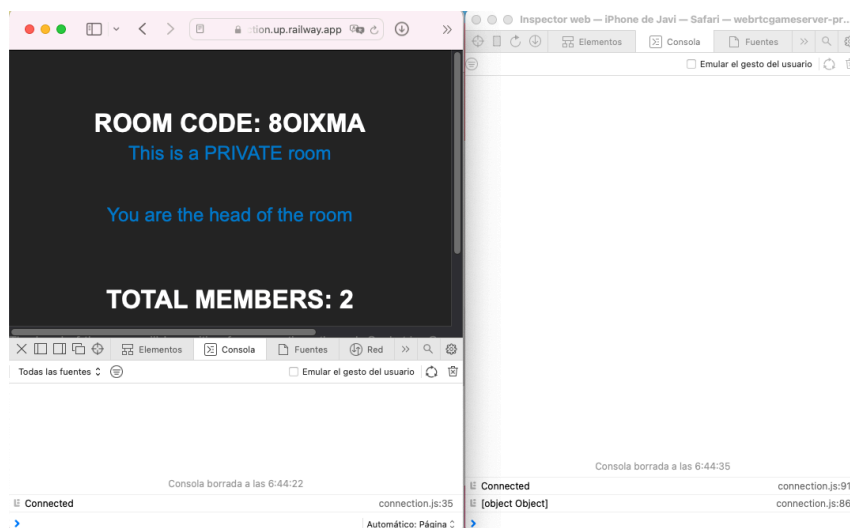


Figura 8: Conexión WebRTC realizada

Una vez comienza la partida también comenzará el bucle de juego. Dentro de este bucle, el desarrollador deberá decidir cuando envía la información desde los clientes a la cabeza de sala y cuando la cabeza contesta a los clientes. Sin embargo, entre estos dos sucesos, se debe utilizar la variable `window.isHead` en el lado del cliente para que, solamente la cabeza, ejecute la lógica necesaria para actualizar un estado de juego que pueda enviar al resto de clientes.

Cabe destacar que, si el desarrollador que utilice esta herramienta está de acuerdo con la arquitectura de comunicación y los detalles de la implementación original, este solo deberá añadir un servidor TURN a la configuración del lado del cliente usada para abrir las conexiones Peer to Peer. Una vez abierto a la red, el programa ya está preparado para funcionar gracias a dos métodos a disposición del usuario de la herramienta: `sendLocalUodate` y `broadcast`. Estas funciones se encargan de la comunicación cabeza-clientes y cliente-cabeza.

6. Resultados y conclusiones

Resultados

Tras el proceso de desarrollo, se ha obtenido una herramienta disponible en GitHub que cumple con los objetivos marcados. Evidentemente, es un proyecto ampliable y con espacio de mejora, sin embargo, ha proporcionado unos resultados adecuados.

El repositorio mencionado contiene un enlace a la web en la que se puede probar la demo generada por el código del propio repositorio. Esta demo permite conectar varios dispositivos siguiendo la estructura descrita durante el desarrollo del proyecto. Sin embargo, al tratarse de una sencilla demo, las conexiones se ven limitadas a dispositivos que se encuentran en la misma red. La idea es que otros desarrolladores puedan clonar el repositorio en cuestión para usar dicha demo como la base de sus proyectos y modifiquen su funcionalidad con servidores TURN, para que las conexiones sean posibles a través de internet. Junto a esta, otras modificaciones pueden ser realizadas para ajustar la arquitectura de comunicación, u otros aspectos del proyecto, a las necesidades del mismo. Un ejemplo claro podría ser el uso de librerías ya incluidas como Socket.io para crear salas de espera en tiempo real en las que los jugadores puedan ver como otros usuarios entran en la sala.

Los resultados proporcionados por la herramienta desarrollada también pueden ser relativamente cuantificados. El sistema no ha sido utilizado en ningún escenario real todavía, sin embargo, durante el proceso de desarrollo se hicieron varias pruebas y experimentos que permiten observar si las premisas por las cuales se utilizaron las tecnologías seleccionadas estaban en lo cierto.

Al fin y al cabo, el servidor tiene 2 funciones: responder a las peticiones HTTP de los clientes cuando buscan una sala y asistir en el proceso de señalización cuando se establece la conexión Peer to Peer. Una vez formada dicha conexión, el trabajo del servidor es casi inexistente. De hecho, solamente intervendrá en el caso de que se creen nuevos candidatos ICE para la conexión WebRTC. Como ya se demostró previamente, un servidor HTTP es considerablemente más lento y costoso que uno basado en Web Sockets, sin embargo, a esto se le debe añadir que, en el proyecto desarrollado, el tráfico de la partida no pasa por el servidor, ni siquiera mediante el socket. Esto demuestra con certeza que tanto el uso, los requisitos y el coste de los servidores utilizados, es incluso más bajo que el de un servidor Web Socket, tanto en el caso de alquiler, como en el uso de alguna máquina propia. Por ejemplo, las famosas Raspberry Pi.

La plataforma de hosting utilizada para las pruebas realizadas durante el desarrollo proporciona unos gráficos de uso que soportan estos resultados. Los dos recursos limitados dentro de un servidor son la CPU

y la memoria RAM. Los cambios en la memoria RAM son despreciables debido a su reducida escala y a que en ningún momento superó unos valores considerables. Sin embargo, el uso de la CPU en el momento de probar el sistema de salas mediante peticiones y respuestas HTTP, es significativamente mayor que en el momento de probar los sockets y la comunicación por WebRTC.

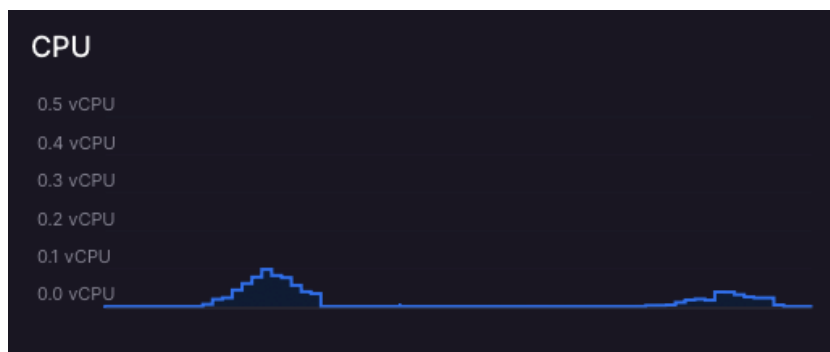


Figura 9: Uso de recursos WebRTC Game Server

Conclusiones

Gracias a la investigación y el desarrollo de la herramienta tratada en esta memoria, se puede concluir que existen infinitas rutas a seguir a la hora de afrontar el back end de un videojuego multijugador. Este proceso presenta ciertas dificultades y cada una de ellas puede ser tratada mediante el uso de diferentes técnicas. Sin embargo, cada día se lanzan nuevas tecnologías y frameworks que asisten en dichas funciones, simplificándolas y haciéndolas más accesibles a desarrolladores indie. Entre dichas tecnologías, existen las herramientas basadas en comunicación en tiempo real, las cuales siguen un paradigma que aporta ventajas durante el desarrollo.

También cabe destacar que, como toda herramienta, las tecnologías mencionadas, incluyendo la desarrollada en este proyecto, pueden ser utilizadas de diversas formas. Otro aspecto que tiene un gran efecto en la eficacia y la eficiencia del resultado final de un desarrollo de estas características, es la arquitectura de comunicación utilizada, el modelo de datos y el tratamiento de los mismos. Esto hace que, aún cuando se puede reducir la dificultad del proceso investigado, el desarrollo de servidores sigue dependiendo en gran medida de los específicos del producto final y del enfoque tomado por el desarrollador.

7. Post mortem y líneas de futuro

El desarrollo de este trabajo y del juego Save the King ha sido un ejercicio didáctico en muchos aspectos, sin embargo, en el proceso también se cometieron varios errores.

En primer lugar el equipo era pequeño y estaba formado por solamente dos programadores, ningún artista. Esto tuvo consecuencias a la hora de llegar a las fechas fijadas, sin embargo, otro factor a tener en cuenta fue una mala organización del propio equipo. Cada miembro se centró en su parte individual para luego poner todo en común y realizar el juego final. Sin embargo, no se llegó a ese punto por falta de tiempo y se acabó con varios proyectos separados con distintas componentes del juego en cada uno, es decir, un

proyecto con el back end, otro con un experimento de Inteligencia artificial, otro proyecto con un blockout y las colisiones pero ningún proyecto conjunto terminado.

Debido a la composición del grupo, el arte nunca pudo llegar a terminarse, sin embargo, tampoco se pudo implementar toda la lógica del juego, las condiciones de victoria ni la inteligencia artificial. Incluso la parte más avanzada, el back end, tuvo fallos que no se llegaron a arreglar.

En un futuro, sería interesante contar con más compañeros, especialmente artistas, para poder acabar el juego. Si repartimos el trabajo de arte, los programadores podríamos terminar aspectos como la inteligencia artificial del juego o los fallos del back end. De hecho, el sistema de back end se podría mejorar bastante modificando la arquitectura de comunicación implementada en la herramienta WebRTC Game Server, cambiando la manera de identificar a los miembros de las salas para utilizar JSON Web Token o añadiendo un servidor TURN público para que se pueda jugar desde fuera de la misma red.

8. Anexos

En la propia carpeta en la que se encuentra esta memoria existe otro directorio llamado “Anexos de programación”. En esa carpeta se encuentran los siguientes documentos de interés para el proyecto:

- Arquitectura de comunicación del proyecto: al inicio del proyecto se realizó un esquema de lo que se creía que acabaría siendo la arquitectura de la herramienta desarrollada. Actualmente, y ya con el proyecto terminado, habría que añadir que la base de datos no fue necesaria. De todas formas, es posible que deba ser implementada para ciertos proyectos de juego específicos. Sin embargo, la idea era desarrollar un sistema para crear la base de un servidor, es decir, esto siempre será extensible por el desarrollador del juego para el que se use esta herramienta.

https://drive.google.com/file/d/1Aubl-dOo1-nXfcB1MfCqXDK-vpxKcWGk/view?usp=share_link

- Código del proyecto: también existe una carpeta con el nombre de la herramienta. En esta carpeta se encuentra el propio proyecto con la totalidad del código. Estos archivos también se encuentran en un repositorio de GitHub.

<https://github.com/Javlregui/WebRTCgameServer>

No se adjunta un manual ya que no resulta necesario. La demo contiene todas las instrucciones para su uso, mientras que la implementación en otros proyectos se realiza mediante la clonación del repositorio. Dentro del código del mismo, todos los componentes están comentados de manera que su uso queda claro y existe un archivo con las funciones disponibles para enviar mensajes dentro de las salas. El resto queda abierto a el uso que le de el desarrollador que haga uso de la herramienta.

9. Bibliografía

- Iregui, J. (2023). WebRTCgameServer. GitHub.
<https://github.com/JaviIregui/WebRTCgameServer>
<https://webrtcgameserver-production.up.railway.app>
- Arcenio, C., &Gómez, Y. (2008). Arquitectura en red para la confección de videojuegos multijugadores. Universidad de las Ciencias Informáticas. Recuperado de
https://repositorio.uci.cu/jspui/bitstream/ident/TD_1565_08/1/TD_1565_08.pdf
- Espinosa, A. (2018). Concurrido y sin servidores: propuesta arquitectural para un videojuego multijugador en línea utilizando tecnologías de cómputo en la nube. Recuperado de
<https://repositorio.uniandes.edu.co/handle/1992/34962>
- Luecke, D. (27 de enero de 2018). HTTP vs Websockets: A performance comparison. feathersjs.com.
<https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77>
- What Socket.IO is not. Socket.IO. Recuperado de
<https://socket.io/docs/v4/>
Fecha de última modificación: 14 de julio de 2023
Fecha de acceso 2 de agosto de 2023
- Martín, F. (2016). Desarrollo de un sistema de comunicaciones basado en WebRTC. Recuperado de
<https://e-archivo.uc3m.es/handle/10016/27318>
- Acerca del navegador Tor. Tor Project. Recuperado de
<https://tb-manual.torproject.org/es/about/>
Fecha de acceso: 4 de agosto de 2023
- Weins, K. (28 de noviembre de 2016). AWS vs Azure vs Google Cloud Pricing: Compute Instances. flexera.com.
<https://www.flexera.com/blog/cloud/aws-vs-azure-vs-google-cloud-pricing-compute-instances/>
- What is Colyseus?. (27 de enero de 2018). Colyseus Multiplayer Framework. Recuperado de
<https://docs.colyseus.io/>
Fecha de acceso: 8 de agosto de 2023
- Lopez, R. (mramonlopez). (2019). game-room-server. npmjs.com.
<https://www.npmjs.com/package/game-room-server>
- aheckmann. (2010). node-gameroom. GitHub.
<https://github.com/aheckmann/node-gameroom>