JUEGO DE CARTAS CON MINI-JUEGOS Y REALIDAD AUMENTADA UTILIZANDO DISPOSITIVOS MÓVILES

Jordi Santiago Provencio

PROYECTO FINAL DE CARRERA

CONTENIDO

Motivación 2
Descripción del proyecto 3
Explicación del juego 5
Biribit 7
- Restricciones y decisiones 7
- Arquitectura y diseño 9
- Protocolo 11
- Servidor 16
- Cliente 19
- Programa de pruebas 26
- Cliente de Biribit para Unity 30
- Creando el proyecto 32

Configurando la Raspberry Pi 35
- Configurando el punto de acceso 35
- Haciendo del servidor de Biribit un servicio 36

Desarrollo del juego en Unity3D 39
- Estado del juego 39
- Serialización de las operaciones 41
- Creación del juego 41

Integración de Vuforia 45
Planificación y análisis económico 48
- Planificación 48
- Análisis económico 50

Conclusiones y trabajo futuro 51

MOTIVACIÓN

En mi trabajo y por cuenta propia, he trabajado con muchas tecnologías: realidad virtual, gráficos por computador, redes para videojuegos, dispositivos móviles, etcétera. Cuando en su momento me planteé qué trabajo realizar como proyecto de fin de carrera, me propuse combinar todos estos conocimientos y aplicarlos a mi afición por los juegos de mesa tradicionales.

Concretamente, mi objetivo ha sido crear un juego de cartas que es tradicional, en el sentido de que se tiene que jugar presencialmente con cartas físicas. De este modo, la capacidad de los participantes para captar las intenciones de los demás según sus reacciones juega un papel importante (tal y como ocurre en muchos juegos de cartas, como por ejemplo el póker). Por otro lado, el juego tiene también un componente no tradicional, en el sentido de contar con el soporte de dispositivos móviles, que aparte de organizar y gestionar el juego, sirven para realizar acciones sobre este de forma oculta, y muestran información selectiva a los jugadores. En particular, he tratado de incorporar tecnologías de realidad aumentada, que permiten al usuario interaccionar con elementos físicos por medio del dispositivo móvil, y mostrar elementos virtuales que dan información adicional.

He tratado de evitar que fuesen necesarios dispositivos de alta gama, ni acceso a internet, y por tanto de diseñar un juego para el que tan solo se requieren las cartas y un móvil sencillo por jugador. De este modo, se puede jugar en cualquier sitio, como en un bar, en el campo o la playa.

DESCRIPCIÓN DEL PROYECTO

Desde el principio se ha decidido desarrollar el juego con Unity3D. Este es un motor de juegos cuya mayor baza es la fácil portabilidad de cualquier proyecto desarrollado a cualquier plataforma soportada, además de que dispone de herramientas muy sencillas para desarrollar juegos. Adicionalmente, se ha incluido Realidad Aumentada para dar el toque de un juego de cartas tradicional.

La Realidad Aumentada (desde ahora, AR de Augmented Reality en inglés) es una técnica de interacción con elementos físicos reales en la cual, mediante una cámara, los elementos son detectados e interpretados, y a su vez permite crear elementos virtuales dentro de esta realidad capturada por la cámara. La utilidad de AR en este proyecto tiene dos partes:

1. Poder interactuar con las cartas de una forma más natural como, por ejemplo, poder ver el contenido oculto de estas con elementos virtuales, o incluso ser usadas enseñándoselas al dispositivo móvil.
2. Poder realizar estos mini juegos a tiempo real en un espacio físico.

Toda la parte de AR se ha desarrollado con el soporte de la librería Vuforia de Qualcomm, que permite muy fácilmente añadir AR a un proyecto de Unity3D y es una de las librerías referentes de AR actualmente. Además, tiene soporte tanto para tener cartas con identificadores únicos, como para reconocer imágenes con patrones genéricos. Esta última funcionalidad es útil para poder crear el mencionado espacio físico donde se realizarán los mini juegos.

Uno de los retos más grandes a superar en este proyecto es la comunicación entre los dispositivos móviles. Para este juego, se ha requerido de una comunicación entre ellos a tiempo real, pues además de ser un juego por turnos, incluye mini juegos a tiempo real que requieren de poca latencia y ancho de banda.

Un buen sistema de comunicación para dispositivos móviles es Wi-Fi. Todos los móviles de hoy en día en el mercado incluyen esta tecnología, y tiene la latencia y ancho de banda necesarios para los requerimientos de este proyecto. Además, permite crear áreas de red local y por lo tanto, no es necesario tener conexión a internet para que los móviles se comuniquen entre ellos.

Existen tecnologías como la llamada Direct Wi-Fi, o Wi-Fi Peer-to-Peer que permiten a los dispositivos encontrarse entre ellos y comunicarse sin un punto de acceso intermedio. Lamentablemente, esta tecnología no está presente en todos los dispositivos móviles, sobre todo porque los únicos dispositivos móviles con dicha tecnología disponible son los Android en sus versiones 4.0 o superiores, y dado que no se ha querido cerrar la puerta a versiones Android más antiguas o incluso iOS en el futuro, se ha optado por usar una alternativa más conservadora.

Para solucionar este problema, y simplemente por simplicidad, se ha elegido crear nuestro propio punto de acceso portable, que a la vez haga de servidor, almacenaje de partidas, y así permita a los jugadores conectar y desconectar sin restricciones. El hardware elegido para esta tarea ha sido una Raspberry Pi.

Raspberry Pi es una placa integrada todo-en-uno del tamaño de una tarjeta de crédito, suficientemente potente como para ser utilizada como un ordenador. Fue creada con el propósito de enseñar a programar en las escuelas.

La razón por la que se ha elegido este hardware para el proyecto ha sido básicamente: por ser transportable, barato (35€ aproximadamente) y suficientemente potente. Juntamente con una batería portátil y un USB Wi-Fi, se ha hecho que este se comporte como un punto de acceso Wi-Fi y de servidor del juego.

Para el software de comunicación entre clientes y el servidor, existen varias opciones. Unity3D tiene hasta dos sistemas de comunicación por IP para hacer juegos multijugador, de los cuales, ninguno permite que se pueda usar una Raspberry Pi como servidor, ni ningún dispositivo similar. Por no decir que en caso de que fuese posible, los binarios ejecutables generados para tal tarea seguramente serían demasiado pesados para la Raspberry Pi.

Dado que se ha decidido que la Raspberry Pi, además de ser el punto de acceso, también sea el servidor, se ha optado por desarrollar un software propio para la comunicación entre dispositivos, que tenga como prioridad hacer que el servidor sea lo más ligero posible y con la posibilidad de funcionar en una Raspberry Pi. A este software lo he llamado Biribit.

Aún a pesar de que mi deseo era que este proyecto pudiese funcionar en cualquiera de las plataformas móviles más populares, entre ellas iOS y Android, se ha tenido que acotar el desarrollo del proyecto únicamente a Android por falta de recursos. Es decir, aún haciendo el proyecto fácilmente portable para iOS, no se ha dispuesto del material necesario que requiere el desarrollo para esta plataforma, dejando esta tarea como trabajo futuro.

Por lo tanto, el proyecto tiene partes claramente diferenciadas, que podemos identificar como proyectos independientes y autocontenidos. El orden cronológico del desarrollo de estos ha sido el siguiente:

1. El software cliente/servidor, Biribit.
2. La configuración de la Raspberry Pi como punto de acceso y servidor.
3. El juego desarrollado en Unity3d.
4. La integración de AR en el juego.

He decidido que estos proyectos serán explicados en apartados por separado porque, en mi opinión, hacen más fácil el seguimiento de esta memoria.

EXPLICACIÓN DEL JUEGO

El juego se llama Divine and Conquer. Las instrucciones del juego son las siguientes:

Los jugadores, unos sacerdotes de un ficticio credo, siguiendo las indicaciones de una vieja profecía llegan a un templo donde supuestamente El Creador inició la creación del mundo. En este templo hay 20 reliquias. El objetivo de los jugadores es encontrar, de entre las 20 reliquias, las 3 oraciones sagradas. Dichas oraciones, según la profecía, conceden al lector el título de El Profeta si se recitan en orden.

Las reglas son sencillas:

- Las cartas representan las 20 reliquias y cada reliquia estará numerada del 1 al 20.
- Al empezar el juego, el contenido de las cartas se distribuye de forma aleatoria por el dispositivo que crea la partida. Por lo tanto, no hace falta barajar las cartas.
- Los jugadores empiezan con 3 cartas en su posesión.
- Los jugadores sólo ven el contenido de las cartas que tienen en su posesión, pero son capaces de ver siempre el número de las cartas que tienen el resto de jugadores.
- Una carta está en juego si la posee un jugador. Las cartas que no están en juego están en la pila de cartas.
- Cuando un jugador roba, puede coger CUALQUIER carta de la pila de cartas.
- El jugador que use las tres oraciones en orden gana.

Las partidas son una sucesión de rondas, y las rondas consisten en turnos, uno por jugador, y ordenados aleatoriamente al principio de cada ronda.

Durante el turno de un jugador, este puede optar por realizar una de estas dos acciones:

1. Intercambiar FÍSICAMENTE una carta de su posesión por otra carta, que llamaremos OBJETIVO, que esté en posesión de otro jugador. El jugador al que se le intercambia la carta no puede negarse. La carta OBJETIVO no podrá ser intercambiada de nuevo hasta la siguiente ronda. Dado que el intercambio es físico/real/material, todos los participantes han sido testigos del mismo.
2. Usar una carta. Esto se gestiona mediante los dispositivos móviles. El resto de jugadores serán conscientes de cuál es la carta utilizada, pero no sabrán, al menos directamente, cual ha sido su efecto. Inmediatamente después de ser usada, se descarta y se roba otra distinta. La posible acción a realizar viene indicada en la carta, y será una de las siguientes:
    a. *Oración primera*: Se lee la primera parte de la oración. Esta carta sólo se puede usar una vez en toda la partida por jugador.
    b. *Oración segunda*: Se lee la segunda parte de la oración. Se puede usar después de que el propio jugador haya usado la carta Oración primera, y solo se puede usar una vez en toda la partida por jugador.

c. *Última oración*: Se lee la última parte de la oración y se gana el juego. Se puede usar después de que el propio jugador haya usado la carta Oración segunda.

d. *Maldición*: Esta carta no tiene uso.

e. *Radar*: Al usar esta carta, suena el dispositivo del jugador que tiene la oración que tú necesitas. Si no hay ningún jugador que tenga tal oración, no suena nada.

f. *Antirradar*: Al usar esta carta, suena el dispositivo del jugador de tu elección como si de un radar se tratase.

g. *Mezclar*: Intercambia el contenido de dos cartas de tu elección que están en juego. Obviamente, los identificadores (físicos) de estas no se ven alterados.

h. *Revelar*: Escoge una carta que está en juego y se te revelará su contenido.

i. *Reflejar*: Se elige una carta en juego que llamaremos ESPEJO y una carta también en juego que llamaremos DESTINO. Cuando una acción recaiga sobre la carta ESPEJO, la acción recaerá sobre la carta DESTINO y la carta ESPEJO perderá su efecto de espejo (por ejemplo, si la carta 8 es espejo de 9 y usamos la carta mezclar sobre la 8 y la 11, se mezclará la 9 con la 11 y la 8 dejará de ser espejo). Si la carta ESPEJO o la carta DESTINO dejan de estar en juego, la carta ESPEJO pierde su efecto espejo.

Al final de la ronda, si algún jugador ha usado una carta oración, se notificará ese hecho (pero no que jugador lo ha hecho) al resto de jugadores, y se especificará que parte de la oración era.

BIRIBIT

Biribit es un sistema de cliente/servidor que permitirá crear la comunicación entre los dispositivos móviles y el servidor. El proyecto consistirá de dos partes:

- El cliente, que será la librería que usará el juego.
- El servidor, que deberá funcionar en la Raspberry Pi.

Este proyecto ha sido con diferencia el que ha supuesto un mayor desafío en todo el desarrollo del proyecto. Aun habiendo trabajado anteriormente con redes, existían ciertas restricciones que complicaban todo el proceso.

RESTRICCIONES Y DECISIONES

1. EL SERVIDOR Y CLIENTE DEBEN FUNCIONAR EN UNA RASPBERRY PI, WINDOWS, LINUX, ANDROID, MACOSX Y IOS.

La Raspberry Pi es un ordenador con 512 MB de RAM y un procesador limitado. Además, debemos tener en cuenta que parte de la RAM ya está ocupada por el sistema operativo. El sistema operativo que se usará para la Raspberry será un Raspbian, un Linux basado en Debian adaptado para la Pi.

Dado que la librería tiene que ser compatible con toda la gama de dispositivos, tener un buen rendimiento y aprovechar eficientemente la memoria, se ha escogido el lenguaje de programación C++, porque permite controlar tanto el rendimiento como el consumo de memoria, y a su vez se puede programar de forma portable (aunque no es precisamente fácil conseguirlo).

2. EL JUEGO NECESITA COMUNICACIÓN A TIEMPO REAL CON LA MÍNIMA LATENCIA.

En una situación genérica, para cualquier proyecto que requiriese de comunicación entre dos aplicaciones cualesquiera, se utilizaría un socket TCP o incluso por petición HTTP. Muy a mi pesar, mi juego no es una aplicación cualquiera. Si este fuese completamente por turnos, se optaría por alguna de las soluciones mencionadas antes, pero al tratarse de un juego que tiene partes en tiempo real, mi elección es UDP.

Podría dedicar páginas y páginas explicando las ventajas y los inconvenientes de UDP respecto TCP. Las conclusiones resumidas son que UDP gana a TCP en latencia, pero TCP gana a UDP en fiabilidad. Sería mentira si dijese que no necesito algunos aspectos de fiabilidad, como garantía de entrega. En mi caso, necesitaré fiabilidad en contados casos y en la mayoría de casos priorizaré la latencia. Por mi experiencia en proyectos anteriores y porque me dio buenos resultados, escogí utilizar RakNet para solucionar este problema.

RakNet es una librería de redes para desarrollo de juegos que permite añadir capas de fiabilidad de forma selectiva y soluciona los problemas más básicos de UDP como simular conexión, fiabilidad, etcétera. Es decir, permite elegir datagrama a datagrama si se quiere garantía de entrega, preservar el orden de envío, etcétera. Además, es compatible con todos los sistemas operativos que necesitamos para este proyecto.

3. EL JUEGO NECESITA TENER UN REGISTRO DE LOS TURNOS.

Las desconexiones son inevitables en un entorno Wi-Fi y móvil. Sería muy lamentable que, por ejemplo, un jugador se separase accidentalmente del punto de acceso Wi-Fi por recibir una llamada y la partida tuviese que volver a empezar porque este se desconectase y perdiera el estado de la partida.

Por lo tanto, desde el inicio del proyecto, se decidió que el transcurso de una partida se registraría como una lista de acciones que se guardasen en un diario dentro del servidor. Así, cualquier persona que se reconectase, podría retomar el juego leyendo este diario hasta el punto donde lo había dejado.

Por lo tanto, se tuvo que crear un protocolo entre cliente y servidor que tuviera en cuenta el concepto de "partida", "acción" y "jugador". Más adelante se explicará en detalle este protocolo.

4. LA LIBRERÍA DEBE SER USADA DESDE UNITY3D.

Como hemos dicho, Unity3D es un motor que permite desarrollar juegos con gran facilidad, y generarlos para distintas plataformas. Esto es cierto siempre que se utilicen las herramientas que ya incluye de serie, pero presenta algunos retos si de alguna manera se desea utilizar una librería externa.

Muchas de las decisiones de diseño de la librería se tomaron única y exclusivamente a raíz de que debía ser usada desde Unity3D. Por ejemplo, la librería tiene su propio hilo de ejecución porque, aunque Unity3D decida que el juego debe parar de ejecutarse por cualquier razón (estar en segundo plano, minimizado, etcétera), la librería tiene que seguir atendiendo a los mensajes de control del protocolo. Esto permite, por ejemplo, que en cualquier momento el jugador pueda bloquear el dispositivo móvil sin temer a desconectarse de la partida.

Pero hay más dificultades: el lenguaje de programación con el que se trabaja en Unity3D es C#. Para hacer uso de librerías externas, por limitación de C# en sí, estas deben ser C y no C++. Por lo tanto, se tuvo que hacer una capa de código en C que usase la librería C++, y una capa C# que usase la capa de C.

Finalmente, uno de los problemas que tenía Unity3D era que si la librería tenía algún fallo como, por ejemplo, de violación de segmento o similar (muy comunes en C y C++), no solo no dejaba depurar el error, sino que el editor de Unity3D decidía cerrarse. Esto nos llevaba al siguiente punto.

5. LA LIBRERÍA TIENE QUE SER FIABLE.

Si un programa está hecho en su totalidad en C# en Unity3D, un fallo puede pasar desapercibido puesto que se genera una excepción y el motor la atrapa para que no afecte al resto de cosas. Al tratarse de una librería de C++ esto no iba a suceder. Un fallo en la librería no solo significaba que se hacía impracticable trabajar con el editor de Unity3D, sino que el juego final también se cerraría inesperadamente, y no es que sea muy agradable arrancar una y otra vez el juego para terminar una partida.

Irónicamente, el escoger C++ con hilos de ejecución paralelos hacía que la probabilidad de tener fallos aumentase. He trabajado en el pasado con paralelismo y aparte de conocer los mecanismos de protección de datos, sé estructurar el código para saber qué hilo de ejecución es propietario de qué datos, dando así un plus a prevención de fallos. Aún a pesar de toda experiencia pasada,

siempre es una buena práctica hacer programas que prueben tu código. Por eso desarrollé un programa que era un entorno de pruebas donde tenía todas las funcionalidades de la librería una por una.

Este programa era una aplicación gráfica hecha con SDL, OpenGL y una librería muy ligera de interfaces graficas llamada IMGUI, orientada al desarrollo de interfaces de depuración. Esta aplicación implementa un chat como herramienta de prueba para las comunicaciones de tiempo real y es capaz de mostrar el contenido del diario de cada partida en hexadecimal o en texto plano.

6. QUEREMOS QUE EL SISTEMA SEA GENÉRICO PARA CUALQUIER JUEGO, Y QUE VARIAS PERSONAS PUEDAN UTILIZAR EL MISMO SERVIDOR PARA DIFERENTES PARTIDAS E INCLUSO DIFERENTES JUEGOS.

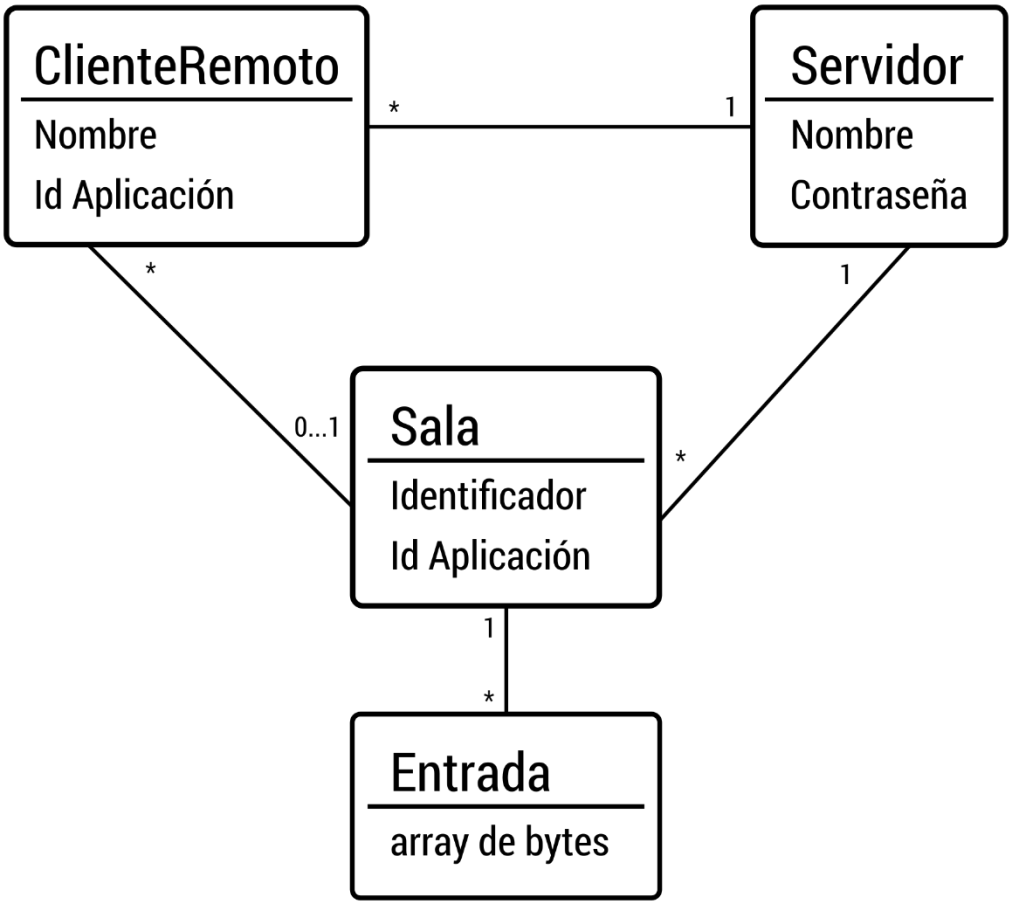
Esta característica no añadió dificultad en la programación, pero sí en el diseño. Esta decisión fue tomada para evitar tener que modificar el código de Biribit mientras se iba desarrollando el juego. De esa manera, se podía trabajar con una versión final.

Básicamente, esta decisión forzó a que el servidor fuese totalmente agnóstico a la lógica del juego, desacoplándolo de procesar los datos de los usuarios y liberándolo así de carga. Dicha decisión supuso matar dos pájaros de un tiro porque, como ya dije antes, la Raspberry Pi no es precisamente el ordenador más potente del mundo.

Por otro lado, esto tiene una desventaja, pues fuerza a que los clientes tengan siempre toda la información de una partida. En nuestro caso se trata de las cartas del resto, o directamente de saber qué contenido hay en las cartas. El juego desarrollado por mí no permitirá este tipo de trampas, pero nadie impide a un tercero malintencionado crear su propia versión del juego permitiendo ver las cartas del resto. Aún así no me preocupa este detalle, dado que ocurre lo mismo que en los juegos de cartas presenciales, donde aún pudiendo hacer trampas no es algo que suela acontecer. De todas maneras, en el caso de que se decidiese desarrollar un juego donde los jugadores se jugaran dinero real, no optaría por hacer el servidor agnóstico.

ARQUITECTURA Y DISEÑO

El gráfico siguiente representa el diseño inicial que se hizo para el servidor. Este resume el concepto del sistema en las siguientes clases.



En este diagrama un cliente es representado como un ClienteRemoto y es identificado por su nombre. El identificador de aplicación indica desde qué aplicación se está usando la librería cliente de Biribit.

La clase Servidor representa como su nombre describe el servidor. Tiene un nombre el cual todos los clientes podrán ver, y una contraseña que los usuarios necesitarán saber para conectarse.

La sala conceptualmente es una partida y es la unidad básica de comunicación para los clientes. Estos solamente pueden comunicarse entre otros clientes de dentro de una sala, y tienen 2 métodos de comunicación:

- MENSAJE DE DIFUSIÓN: Un cliente puede enviar un mensaje de difusión. Este mensaje llegará al servidor e inmediatamente reenviará este mensaje a todos los clientes que formen parte de la sala con la mínima latencia posible. No quedará registrado este mensaje por ningún lado.
- ENTRADA EN EL DIARIO: Un cliente puede enviar un mensaje como entrada del diario. Este mensaje al llegar al servidor es almacenado en el diario de la sala e inmediatamente notifica al resto de clientes que el diario ha sido modificado. Los clientes más tarde podrán consultar. Con este método, cualquier persona que entrara más tarde a la sala podría leer todo el diario.

Como podemos ver, un servidor puede tener más de una sala y por lo tanto, permite que varias personas puedan aprovechar un mismo servidor para hacer más de una partida simultánea.

En la implementación final veremos que las salas tienen huecos. Cuando un cliente entra en una sala, ocupa un hueco. Si un jugador ocupa el hueco número 2 diremos que es el jugador 2. Este concepto se introdujo casi al final del desarrollo de Biribit. Surgió a raíz de que, en caso de desconexión, tenía que haber manera de volver a ser el mismo jugador en una partida. Incluso me di cuenta de que el sistema debía de ser capaz de hacer que dos clientes intercambiaran los papeles (por ejemplo: pasar de observador a participante de la partida). Una vez hecho el sistema de huecos, me di cuenta que podía servir para elegir qué rol se quería ejercer en la partida (por ejemplo: en un juego de ajedrez, se puede especificar que el hueco 1 son las blancas, el hueco 2 las negras y el resto, observadores).

Como hemos indicado en las características, el sistema es agnóstico. Un mensaje es una cadena de bytes que solo los programas cliente saben interpretar. Sin embargo, sería desafortunado que un programa que espera un formato de cadena de bytes determinado recibiese otra cadena de bytes de otro programa distinto. Para evitar estas situaciones, el servidor fuerza que los clientes tengan un identificador de aplicación, y comprueba que estos solo puedan entrar en las salas que tengan el mismo identificador de aplicación.

PROTOCOLO

En general enviar cadenas de bytes entre ordenadores por la red siempre es un problema. No todos los ordenadores son la misma arquitectura, ni interpretan las cadenas de datos de la misma manera. Hay diversos problemas como: endianess, la codificación de los números reales… Por lo tanto, todo programa que envía datos por red, obligatoriamente debe hacer un proceso conocido como Marshalling.

A cuestiones prácticas no hay ninguna diferencia entre los conceptos marshalling y serialización, pero por alguna razón se suele usar el primero en contextos de trasmitir datos, guardar en una base de datos o convertir estructuras de datos equivalentes entre lenguajes de programación.

El objetivo no es profundizar en estas cuestiones, por lo tanto, se ha escogido una librería de terceros que hiciese este trabajo. Hay muchos tipos de librerías para hacer esta tarea, pero debido a que estos mensajes debían enviarse por red, a la hora de buscar se hizo hincapié en la velocidad de serialización y tamaño en bytes de la serialización. Por el otro lado, se sacrificaron otras características como ser editable o legible por una persona (como ocurre en XML o en JSON). La elección final fue Protocol Buffers, también llamado Protobuffers.

Protobuffers es una librería made in Google que es utilizada para los protocolos de comunicación internos entre sus servicios. El objetivo de la librería es agilizar la creación de protocolos, minimizando el tamaño de los mensajes y el tiempo de serialización, pero sin perder la flexibilidad de que un protocolo se pueda expandir, procurando que un programa con una versión de un protocolo más nueva pudiese comunicarse con versiones antiguas del mismo programa.

Para agilizar el tiempo de creación de protocolos, Protobuff tiene un lenguaje propio para definir las estructuras de datos (proto-estructuras). Estas proto-estructuras se pueden compilar y transformar en código C++, de tal manera que permite trabajar con estas estructuras como si de

clases de C++ se tratara. La serialización es rapidísima y es binaria, por lo que pierde legibilidad, pero gana en espacio.

El protocolo de Biribit define que todo datagrama enviado desde el Cliente o desde el Servidor empieza por un identificador de 1 bytes que especifica el tipo de mensaje. En lista siguiente podemos ver los tipos de mensajes seguidos de los datos adicionales con los que se suele enviar el datagrama.

ID	Nombre	Enviado por	Datos adicionales
134	ID_ERROR_CODE	Server	`Proto::BiribitError`
135	ID_SERVER_INFO_REQUEST	Cliente	`---`
136	ID_SERVER_INFO_RESPONSE	Server	`Proto::ServerInfo`
137	ID_SERVER_STATUS_REQUEST	Cliente	`---`
138	ID_SERVER_STATUS_RESPONSE	Server	`Proto::ServerStatus`
139	ID_CLIENT_UPDATE_STATUS	Cliente	`Proto::ClientUpdate`
140	ID_CLIENT_STATUS_UPDATED	Server	`Proto::Client`
141	ID_CLIENT_DISCONNECTED	Server	`Proto::Client`
142	ID_ROOM_LIST_REQUEST	Cliente	`---`
143	ID_ROOM_LIST_RESPONSE	Server	`Proto::RoomList`
144	ID_ROOM_CREATE_REQUEST	Cliente	`Proto::RoomCreate`
145	ID_ROOM_JOIN_RANDOM_OR_CREATE_REQUEST	Cliente	`Proto::RoomCreate`
146	ID_ROOM_JOIN_REQUEST	Cliente	`Proto::RoomJoin`
147	ID_ROOM_STATUS	Server	`Proto:Room`
148	ID_ROOM_JOIN_RESPONSE	Server	`Proto::RoomJoin`
149	ID_SEND_BROADCAST_TO_ROOM	Cliente	`reliability(uint8) + binary data`
150	ID_BROADCAST_FROM_ROOM	Server	`sender_slot(uint8) + binary data`
151	ID_JOURNAL_ENTRIES_REQUEST	Client	`Proto::RoomEntriesRequest`
152	ID_JOURNAL_ENTRIES_STATUS	Server	`Proto::RoomEntriesStatus`
153	ID_SEND_ENTRY_TO_ROOM	Client	`binary data`

ID_ERROR_CODE: El servidor envía un paquete con este código si se produce algún error en el servidor. Los códigos de error son los siguientes:

- WARN_CLIENT_NAME_IN_USE: El cliente ha solicitado cambiar el nombre a uno que ya está en uso.
- WARN_CANNOT_LIST_ROOMS_WITHOUT_APPID: El cliente ha solicitado listar salas sin tener un identificador de aplicación asignado.

- WARN_CANNOT_CREATE_ROOM_WITHOUT_APPID: El cliente ha solicitado crear una sala sin tener un identificador de aplicación asignado.
- WARN_CANNOT_CREATE_ROOM_WITH_WRONG_SLOT_NUMBER: El cliente ha solicitado crear una sala con un numero de huecos para jugadores o bien cero, o bien negativo.
- WARN_CANNOT_CREATE_ROOM_WITH_TOO_MANY_SLOTS: El cliente ha solicitado crear una sala con un numero de huecos para jugadores superior al límite.
- WARN_CANNOT_JOIN_WITHOUT_ROOM_ID: Este mensaje es para depuración. El cliente ha enviado una solicitud de entrar en sala sin especificar la sala.
- WARN_CANNOT_JOIN_TO_UNEXISTING_ROOM: El cliente ha enviado una solicitud de entrar en sala con un identificador de sala no existente.
- WARN_CANNOT_JOIN_TO_OTHER_APP_ROOM: El cliente ha enviado una solicitud de entrar en sala con un identificador de aplicación distinto a la del cliente.
- WARN_CANNOT_JOIN_TO_OCCUPIED_SLOT: El cliente ha enviado una solicitud de entrar en sala y ha especificado un hueco ya ocupado.
- WARN_CANNOT_JOIN_TO_INVALID_SLOT: El cliente ha enviado una solicitud de entrar en sala y ha especificado un hueco que no existe.
- WARN_CANNOT_JOIN_TO_FULL_ROOM: El cliente ha enviado una solicitud de entrar en sala sin especificar hueco, pero todos los huecos de la sala están llenos.

ID_SERVER_INFO_REQUEST: El cliente solicita la información del servidor. Esta petición la puede hacer un cliente sin estar conectado al servidor.

ID_SERVER_INFO_RESPONSE: El servidor responde a ID_SERVER_INFO_REQUEST. La definición de `Proto::ServerInfo` es el siguiente:

```
message ServerInfo
{
	required string name = 1; //nombre del servidor
	required bool password_protected = 2; //protegido con contraseña
	required uint32 max_clients = 3; //número máximo de clientes
	required uint32 connected_clients = 4; //clientes conectadas
}
```

Notación de Protobuffers:

- `message` se comporta igual que una estructura.
- `required` fuerza a que un campo sea obligatorio.

La numeración de los campos es parte del protocolo. Sirve para ampliar los mensajes en el futuro y permitir que aplicaciones con definiciones de los mensajes antiguas funcionen con las nuevas, siempre y cuando los números no colisionen.

ID_SERVER_STATUS_REQUEST: El cliente solicita la lista de clientes remotos conectados en el servidor.

ID_SERVER_STATUS_RESPONSE: El servidor responde a ID_SERVER_STATUS_REQUEST. La definición de `Proto::ServerStatus` es la siguiente:

```
message Client
{
	required uint32 id = 1; //identificador de cliente [1..N]
	required string name = 2; //nombre de cliente
	optional string appid = 3; //identificador de aplicación

	//un cliente se identificará entre la lista de clientes si este campo
	esta a true para él.
	optional bool self = 4;
}

message ServerStatus
{
	repeated Client clients = 1; //lista de clientes del servidor
}
```

Notación de Protobuffers:

- `repeated` indica que es un vector. Puede estar vacío.
- `optional` indica que el valor puede ser nulo

ID_CLIENT_UPDATE_STATUS: El cliente solicita cambiar su nombre, identificador de aplicación en el servidor. La definición de `Proto::ClientUpdate` es la siguiente:

```
message ClientUpdate
{
	optional string name = 1; //nombre nuevo
	optional string appid = 2; //identificador de aplicación nuevo
}
```

ID_CLIENT_STATUS_UPDATED: El servidor responde a ID_CLIENT_UPDATE_STATUS. Envía un `Proto::Client` con los nuevos datos.

ID_CLIENT_DISCONNECTED: El servidor informa a todos los clientes que un cliente se ha desconectado. Envía un `Proto::Client` para especificar que cliente se ha desconectado.

ID_ROOM_LIST_REQUEST: El cliente solicita lista de salas en el servidor.

ID_ROOM_LIST_RESPONSE: El servidor responde a ID_ROOM_LIST_REQUEST. La definición de `Proto::RoomList` es la siguiente:

```
message Room
{
	//identificador de la sala
	required uint32 id = 1;

	//ids de los clientes entrados
	repeated uint32 joined_id_client = 2;

	//numero de entradas en el diario
	optional uint32 journal_entries_count = 3;
}
```

```
message RoomList
{
	repeated Room rooms = 1;
}
```

ID_ROOM_CREATE_REQUEST: El cliente solicita crear una sala nueva en el servidor y entrar en ella. La definición de `Proto::RoomCreate` es la siguiente:

```
message RoomCreate
{
	required uint32 client_slots = 1; //número de huecos para jugadores
	optional uint32 slot_to_join = 2; //hueco que se pide ocupar al crear
}
```

ID_ROOM_JOIN_RANDOM_OR_CREATE_REQUEST: El cliente solicita entrar en una sala en la que haya al menos un hueco libre, y en caso de no haber, crea una sala. Envía un `Proto::RoomCreate` pero sólo se usa en el caso de que se haya creado sala.

ID_ROOM_JOIN_REQUEST: El cliente solicita entrar en una sala. La definición de `Proto::RoomJoin` es la siguiente:

```
message RoomJoin
{
	optional uint32 id = 1; //identificador de sala
	optional uint32 slot_to_join = 2; //hueco que se desea ocupar
}
```

ID_ROOM_STATUS: El servidor cada vez que entra o sale un cliente de la sala, informa del estado de la sala a los clientes que están dentro. Envía un `Proto:: Room`.

ID_ROOM_JOIN_RESPONSE: El servidor responde a ID_ROOM_JOIN_REQUEST. Envía un `Proto::RoomJoin` confirmando la sala y el hueco ocupado.

ID_SEND_BROADCAST_TO_ROOM: El cliente envía un mensaje de difusión a la sala. Se envía un byte que indica la fiabilidad que se le debe aplicar al datagrama, seguido de la cadena de bytes que se desea enviar. La fiabilidad está controlada por RakNet, así este valor simplemente es indicado a la librería. Este es el listado nivel de fiabilidad permitido para un paquete difusión:

- UNRELIABLE: Se envía sin importar si llega a los destinatarios. Este tipo de mensaje es útil para juegos que están enviando constantemente el estado de los jugadores. En este tipo de juegos, los mensajes antiguos perdidos dejan de cobrar importancia porque se sabe seguro que se recibirá un nuevo mensaje con un estado más actualizado.
- RELIABLE: Se envía el mensaje y se guarda en un buffer esperando confirmación de los destinatarios, si pasa un tiempo sin llegar la confirmación se reenvía el mensaje, por lo que implica que el mensaje no se borra hasta que no se recibe la confirmación. *Observación*: Normalmente, en una conexión TCP, cuando se supera el máximo de datos pendientes por enviar, el programa es bloqueado en la llamada de enviar. Usando RakNet al no ser bloqueante, puede haber riesgo de que si se envían demasiados datos sin confirmación se llene demasiado la memoria. Esto provoca que en muchas situaciones se deba tener cuidado pues se podría desbordar la memoria fácilmente.

- RELIABLE_ORDERED: Es lo mismo que el anterior, pero el receptor en caso de recibir un mensaje N antes que un mensaje N-1 o con el identificador menor, lo retiene en un buffer hasta que recibe todos los posteriores para entregarlos al usuario a la vez.

ID_BROADCAST_FROM_ROOM: El servidor envía el mensaje de difusión enviado por un cliente de la sala a todos los clientes de la sala, incluido el emisor. Se envía un byte que indica desde que hueco de la sala ha sido enviado, seguido de la cadena de bytes del mensaje.

ID_JOURNAL_ENTRIES_REQUEST: El cliente solicita al servidor que se le envíe el listado de entradas del diario indicados a continuación. La definición de `Proto::RoomEntriesRequest` es la siguiente:

```
message RoomEntriesRequest
{
	repeated uint32 entries_id = 3;//identificadores de las entradas
}
```

ID_JOURNAL_ENTRIES_STATUS: El servidor envía este mensaje a todos los clientes en caso de que un cliente haya añadido una entrada en el diario, pero no envía los datos de ninguna entrada. También envía este mensaje a un solo cliente cuando este cliente ha enviado un ID_JOURNAL_ENTRIES_REQUEST solicitando un listado de entradas del diario. En tal caso en la `entries` de la estructura `Proto::RoomEntriesStatus` estarán las entradas solicitadas. Estas son las estructuras implicadas:

```
message RoomEntry
{
	required uint32 id = 1; //índice del diario que ocupa la entrada
	required uint32 from_slot = 2; //índice del hueco desde donde se envió
	required bytes entry_data = 3; //datos de la entrada
}

message RoomEntriesStatus
{
	optional uint32 room_id = 1;//identificador de la sala
	optional uint32 journal_size = 2;//numero de entradas de la sala
	repeated RoomEntry entries = 3; //entradas solicitadas por el cliente
}
```

ID_SEND_ENTRY_TO_ROOM: El cliente envía una entrada nueva para el diario de la sala. La cadena de bytes a continuación es el mensaje.

SERVIDOR

Ya se ha hablado del protocolo y del diseño del servidor, pero ¿qué es el servidor? El servidor es un ejecutable de línea de comandos. Está hecho así porque es la manera más fácil para convertirlo en un servicio de sistema para Linux. He aquí lo que muestra el ejecutable del servidor con el parámetro para mostrar la ayuda del programa:

```
$ ./BiribitServer.exe --help

USAGE:
```

```
   d:\Biribit\BiribitServer.exe  [-m <maxclients>] [-w <password>] [-p
                                 <port>] [-n <name>] [--] [--version] [-h]


Where:

   -m <maxclients>,  --maxclients <maxclients>
     Max clients can connect

   -w <password>,  --password <password>
     Server password

   -p <port>,  --port <port>
     Port to bind

   -n <name>,  --name <name>
     Server name shown

   --,  --ignore_rest
     Ignores the rest of the labeled arguments following this flag.

   --version
     Displays version information and exits.

   -h,  --help
     Displays usage information and exits.


   Biribit server
```

El sistema de mostrar ayuda y capturar parámetros por línea de comandos es facilitado por TCLAP. Esta es una librería en C++ que solo requiere de cabeceras, por lo que la integración solo requiere de hacer un `#include`.

El servidor una vez arrancado muestra un registro detallado de los eventos que ocurren durante la ejecución.

```
D:\BuD\Documents\Biribit\build\x64\bin\MinSizeRel\BiribitServer.exe
[13:39:04] Starting server. Server port: 8287
[13:39:05] Failed to start dual IPV4 and IPV6 ports. Trying IPV4 only.
[13:39:05] IPV4 only success.
[13:39:05] Network interface(s) detected: 1
 1. 192.168.1.129 (LAN=0)

[13:39:05] Server "Shenika" running successfully
[13:39:10] New client(1) "Kesha23" connected from 127.0.0.1|58980.
[13:39:17] Client(1) "Kesha23" changed name to "ClientName".
[13:39:17] Client(1) "ClientName" changed appid to "app-client-test".
[13:39:19] Created room 1 for the app app-client-test.
[13:39:19] Client (1) "ClientName" joins room 1.
[13:39:33] New client(2) "Raymonde47" connected from 127.0.0.1|56157.
[13:39:44] Client(2) "Raymonde47" changed name to "OtherClient".
[13:39:44] Client(2) "OtherClient" changed appid to "app-client-test".
[13:39:48] Client (2) "OtherClient" joins room 1.
[13:39:58] Client 127.0.0.1|58980 disconnected.
[13:39:58] Client (1) "ClientName" leaves room 1.
[13:40:00] Client (2) "OtherClient" leaves room 1.
[13:40:00] Room 1 is empty. Closing room.
[13:40:02] Client 127.0.0.1|56157 disconnected.
[13:40:17] New client(1) "Ardella22" connected from 127.0.0.1|63866.
[13:40:24] Client 127.0.0.1|63866 disconnected.
```

Como hemos visto antes en los parámetros de arranque, se permite al servidor seleccionar nombre, contraseña, puerto y más cosas. La ventana adjuntada justo arriba muestra la ejecución de un servidor sin ningún parámetro, por lo tanto, todos los parámetros de arranque son opcionales.

Aun siendo el nombre opcional como parámetro, es requerido para el funcionamiento del servidor. En caso de no proporcionarse uno, se escoge uno aleatorio de una lista de 50 nombres. En el ejemplo de arriba podemos ver como se asigna el nombre de “Shenika”.

RakNet es el encargado de gestionar las conexiones. Al iniciarse, RakNet solicita un puerto, un número de conexiones máximo y una contraseña. Si no se facilita ninguno de estos, asigna por defecto el máximo de 50 conexiones, el puerto 8287 y ninguna contraseña.

Siguiendo con la dinámica, vemos en la imagen que se conecta un cliente. Cuando un cliente se conecta, se le asigna un nombre aleatorio de una lista de 50 nombres y se le concatena un número de 2 cifras calculado a partir de la hora en que se ha conectado. El nombre de un cliente forzosamente es único. El identificador de la aplicación se inicializa como nulo. Todos los clientes conectados son notificados de esta nueva conexión y el cliente conectado es notificado de su nombre, y de la lista de clientes ya conectados.

Un cliente puede solicitar cambiarse el nombre y el identificador de aplicación enviando ID_CLIENT_UPDATE_STATUS como se ha explicado en el apartado de protocolo. Como hemos destacado antes, los nombres son únicos, por lo que si el nombre solicitado está en uso, el servidor resuelve el conflicto añadiendo un número al final del nombre [1, 2, 3…] hasta que encuentre un

nombre que no entre en conflicto. Vamos a ver una muestra de esta resolución de conflicto en la siguiente imagen.

```
D:\BuD\Documents\Biribit\build\x64\bin\MinSizeRel\BiribitServer.exe
[14:06:32] Starting server. Server port: 8287
[14:06:32] Failed to start dual IPV4 and IPV6 ports. Trying IPV4 only.
[14:06:32] IPV4 only success.
[14:06:32] Network interface(s) detected: 1
 1. 192.168.1.129 (LAN=0)

[14:06:32] Server "Keren" running successfully
[14:06:35] New client(1) "Tandy42" connected from 127.0.0.1|63996.
[14:06:40] New client(2) "Kesha6" connected from 127.0.0.1|63997.
[14:06:51] New client(3) "Tiffani74" connected from 127.0.0.1|52982.
[14:07:04] Client(1) "Tandy42" changed name to "ClientName".
[14:07:04] Client(1) "ClientName" changed appid to "app-client-test".
[14:07:13] Client(2) "Kesha6" changed name to "ClientName1".
[14:07:13] Client(2) "ClientName1" changed appid to "app-client-test".
[14:07:18] Client(3) "Tiffani74" changed name to "ClientName2".
[14:07:18] Client(3) "ClientName2" changed appid to "app-client-test".
```

Tres clientes se conectan a un servidor y los tres solicitan llamarse “ClientName”. Solo el primero adquiere el nombre solicitado, sin embargo, los dos restantes adquieren “ClientName1” y “ClientName2”. Una vez que un cliente se cambia el nombre, el resto de clientes son notificados.

Respecto a las salas, todas tienen el nombre de aplicación del cliente que crea la sala. Los clientes solo pueden listar las salas con el mismo nombre de aplicación.

Lógicamente, si un cliente intenta listar salas, crear una sala o unirse a una sala y no tiene nombre de aplicación, no podrá. Además, el servidor le notificará mediante un error que la solicitud que ha hecho ha provocado un error.

Cuando un cliente crea una sala, especifica el número de huecos y se une a la sala automáticamente. Cuando los huecos están todos ocupados, no se permitirá entrar más clientes a la sala. El límite de huecos es 255 porque el protocolo fuerza a representar el índice de hueco en 1 byte.

Si un cliente no tiene nombre de aplicación no puede crear salas. Un cliente solo puede estar en una sala a la vez. Una vez dentro de una sala, los clientes unidos a esta serán notificados con un ID_ROOM_STATUS cada vez que alguien entra o sale de la sala.

CLIENTE

Una vez descrito el servidor, nos centramos en el Cliente. Este no será una aplicación sino una librería que será usada por el juego.

Dentro de la librería, hay la clase `Biribit::Cliente` que representara el estado de un cliente. Con una instancia de esta clase se pueden realizar las operaciones descritas en el protocolo. Cada instancia arranca un hilo de ejecución que lleva a cabo de forma asíncrona las operaciones. Esto es así porque muchas de estas operaciones requieren de la respuesta del servidor y pueden tardar más de lo que tarda el juego en dar una iteración del bucle principal. No sería practico ni sensato bloquear el programa durante un tiempo muy significativo. Además, el cliente debe atender a paquetes que le llegan desde el servidor sin que tenga operaciones pendientes, y con tal de agilizar la comunicación es necesario que el cliente atienda a estos paquetes tan pronto como sea posible desde su hilo de ejecución.

Por lo tanto, como podremos observar más adelante, ninguna de las operaciones devuelve resultado inmediato. Para cubrir esta necesidad de devolver resultados (como, por ejemplo, informar de que se ha podido conectar al servidor) se usan eventos. El usuario, a cada iteración de la lógica del juego podrá consultar esta cola de eventos, extrayéndolos uno por uno hasta que se vacíe.

El método para extraer eventos de esta cola está en la clase cliente, y es la siguiente:

```
std::unique_ptr<Event> DesencolarEvento();
```

Nota: Un `unique_ptr` es una estructura de la librería estándar de C++ que encapsula un puntero. Este puntero no se puede copiar y una vez que desaparece del alcance se libera su memoria.

La función `DesencolarEvento` retorna un puntero al evento que está el primero en la cola y lo desencola. Cuando la cola esté vacía, se devolverá un puntero nulo.

Evento es la clase base y tiene un identificador que indica qué tipo de evento es.

```
struct API_EXPORT Event
{
  EventId id;

  Event();
  Event(EventId);
  virtual ~Event();
};
```

Cada tipo de evento hereda de la clase base y asigna en su constructor el tipo. Algunos eventos añaden atributos extra, otros son simplemente para notificar algún suceso. Los identificadores de eventos son los siguientes:

```
enum EventoId
{
  EVENT_NONE_ID = 0, // No tiene uso
  EVENT_ERROR_ID, // El servidor ha enviado un error
  EVENT_SERVER_LIST_ID, // La lista de servidores se ha actualizado
  EVENT_CONNECTION_ID, // El cliente se ha conectado o desconectado
  EVENT_SERVER_STATUS_ID, // El servidor envía información del estado
```

```
  EVENT_REMOTE_CLIENT_ID, // Un cliente remoto ha cambiado su estado
  EVENT_ROOM_LIST_ID, // El servidor ha enviado la lista de salas
  EVENT_JOINED_ROOM_ID, // El cliente ha entrado o salido de una sala
  EVENT_BROADCAST_ID, // Se ha recibido un paquete de difusión
  EVENT_ENTRIES_ID, // Se han añadido o se han recibido entradas
};
```

Una vez identificado el tipo de evento, simplemente se reinterpreta el puntero como el tipo de evento que le corresponde. Ahora, introduciremos los métodos con los que se puede trabajar con el cliente y comentaremos qué eventos se generan por cada operación.

Habitualmente, una conexión comienza proporcionando una dirección IP o un nombre de dominio. Un usuario de un juego en general espera que el juego le proporcione algún método para conectarse y empezar una partida fácilmente. Normalmente los juegos vienen incluidos con una dirección a un servidor master. Este servidor master proporciona listados de servidores activos donde los clientes se pueden conectar.

Una vez que el cliente tiene un listado de servidores, este solo debe comunicarse con los servidores uno a uno para obtener datos como latencia, nombre, clientes conectados, etcétera.

El cliente de Biribit tiene implementada una lista de servidores, pero no usa ningún servidor master. Sin embargo, sí puede listar los servidores que estén en red local. Esta característica es muy útil para el proyecto porque al estar en la misma red Wi-Fi, podemos determinar cuál es la dirección del servidor de la Raspberry Pi.

Para encontrar servidores en red local, el cliente de Biribit envía un paquete PING a la dirección de difusión de la red (255.255.255.255) a la espera de que servidores en la misma red respondan con un paquete PONG. El paquete PING no es un ICMP, sino que es parte del protocolo de RakNet y se hace mediante un paquete UDP con el identificador de paquete PING. Lo mismo se aplica a PONG.

Por la naturaleza de UDP, RakNet permite comunicación sin que haya conexión. El concepto de conexión no es más que una herramienta de RakNet para tener control si se ha perdido el enlace con un cliente. Funciona a base de tener registro sobre cuándo fue la última vez que un cliente conectado envió un paquete, y enviar un paquete PING en caso de que peligrosamente se acerque el tiempo máximo para dar por desconectado a un cliente. Si un cliente supera el tiempo máximo, se notifica que tal cliente se ha desconectado.

Como ya se ha dicho antes, un cliente no tiene por qué estar conectado a un servidor para comunicarse con él. Sin embargo, se ha limitado en el protocolo de Biribit para que solo se pueda usar parte del protocolo, en concreto, la parte para consultar la información del servidor.

El procedimiento para encontrar servidores es la siguiente:

1. El cliente envía un paquete PING a la dirección de difusión (255.255.255.255).
2. Un servidor en red local recibe el paquete PING y envía un paquete PONG al cliente.
3. El cliente envía un paquete ID_SERVER_INFO_REQUEST al servidor.
4. El servidor responde enviando un paquete ID_SERVER_INFO_RESPONDE al cliente.

El cliente añade el servidor a la lista de servidores y genera un evento de tipo `EventServerList` para notificar que se han encontrado servidores.

Los métodos de la clase cliente que están relacionados con esta funcionalidad son los siguientes:

```
void EncontrarServidoresEnLAN(unsigned short puerto);
void LimpiarListaDeServidores();
void ActualizarServidoresListados();
```

`EncontrarServidoresEnLAN` generará un evento de tipo `EventServerList` en caso de encontrar servidores satisfactoriamente. Este es un evento vacío, y simplemente notifica que se ha actualizado la lista de servidores. Para obtener la lista de servidores hay que utilizar el método siguiente:

```
future_vector<ServerInfo> ObtenListaServidores();
```

NOTA: Por la naturaleza asíncrona de la librería, cuando se solicita según qué información a la clase cliente, esta no devuelve el resultado inmediatamente. En su defecto, se devuelve una estructura `future_vector`, la cual no es más que un alias a `future<vector<Tipo>>`. Una `future` es una estructura que almacena un resultado que puede que tarde tiempo en ser asignado. Tan pronto como el hilo de ejecución del cliente atienda a la petición, la estructura confirmará que el resultado ya está disponible y podrá ser accedido. El juego deberá almacenar esta estructura, y a cada iteración consultará si ya ha sido asignado el resultado.

La definición de `ServerInfo` es la siguiente:

```
struct ServerInfo
{
  string name; //nombre del servidor a mostrar al usuario
  string addr; //dirección IP o nombre de dominio
  uint32_t ping; //latencia
  unsigned short port; //puerto
  bool passwordProtected; //está protegido con contraseña?
  unsigned int max_clients; //máximo de clientes conectados
  unsigned int connected_clients; //clientes conectados actualmente
};
```

Una vez que ya se dispone de una dirección IP o nombre de dominio para conectarse, la clase cliente puede usar los siguientes métodos para manejar la conexión:

```
void Conectar(const char* addr, unsigned short puerto, const char* password);
void Desconectar(Connection::id_t id);
```

En ambas operaciones, todos los parámetros son opcionales y tienen valores por defecto. Si no se asigna una dirección, se intentará conectar a la dirección local. Si no se asigna puerto, se intentará conectar al puerto 8287. Si no se asigna contraseña, se realizará una conexión sin contraseña.

El evento generado con estos métodos es `ConnectionEvent`, y estos son los atributos:

```
struct ConnectionEvent : public Event
{
  enum EventType {
    TYPE_NEW_CONNECTION, // se ha producido una nueva conexión
    TYPE_DISCONNECTION, // el cliente se ha desconectado
```

```
    TYPE_NAME_UPDATED // se ha actualizado la información de la conexión
  };

  EventType type;
  Connection connection;
};
```

La estructura Connection contiene el identificador de conexión, junto con más información:

```
struct Connection
{
  using id_t = std::uint32_t;

  id_t id; // identificador de la conexión
  std::string name; // nombre del servidor
  std::uint32_t ping; // latencia (se va actualizando)
};
```

También se puede obtener las conexiones con el método siguiente:

```
future_vector<Connection> ObtenConexiones();
```

Cuando un cliente se conecta, el servidor asigna al cliente un nombre provisional y es enviado al cliente. El cliente recibe esa información y genera un evento de tipo RemoteClientEvent.

```
struct RemoteClientEvent : public Event
{
  enum EventType {
    TYPE_CLIENT_UPDATED,
    TYPE_CLIENT_DISCONNECTED
  };

  EventType type;
  Connection::id_t connection;
  RemoteClient client;
  bool self;
};
```

En el caso de que el servidor te informe de tu nombre provisional, `type` será TYPE_CLIENT_UPDATED y el booleano `self` será cierto. Este evento también sirve para informar sobre cambios de nombre, identificador de aplicación de otros clientes, o incluso para informar de desconexiones. Los datos de la estructura `RemoteClient` son los siguientes:

```
struct RemoteClient
{
  using id_t = std::uint32_t;


  id_t id;
  string name;
  string appid;
};
```

La clase cliente internamente tiene una copia del estado de los clientes. Cada vez que el servidor va notificando, esta va actualizando para que coincida con el estado del servidor. Cuando la clase cliente se conecta, envía un paquete de tipo ID_SERVER_STATUS_REQUEST pidiendo una lista inicial de los estados de los clientes. El servidor responde con un paquete de tipo ID_SEVER_STATUS_RESPONSE y el cliente genera un evento de tipo `ServerStatusEvent`:

```
struct ServerStatusEvent : public Event
{
  Connection::id_t connection;
  std::vector<RemoteClient> clients;
};
```

La información de todos los clientes es pasada al usuario de la librería cliente, pero la clase cliente sigue manteniendo una copia. Se puede solicitar una copia usando el siguiente método de la clase cliente:

```
future_vector<RemoteClient> ObtenClientesRemotos(Connection::id_t id);
```

Una vez establecida la conexión, lo siguiente que debería hacer la aplicación usuaria de la librería cliente es cambiar el nombre del cliente y el identificador de la aplicación.

El identificador de aplicación está definido a nivel de aplicación. La aplicación tiene que asignar a sus clientes un identificador único para que no se mezcle con clientes de otra aplicación. El identificador de aplicación se puede usar incluso para separar los clientes de diferentes versiones de un mismo juego. En el caso de este juego, el identificador usado es “divine-and-conquer-1”.

Para poder cambiar tanto el nombre como el identificador de aplicación se debe usar el método siguiente.

```
void AsignarParametrosCliente(Connection::id_t id, const ClientParameters&
      parametros);
```

Si el cambio es satisfactorio, recibiremos un evento del tipo `RemoteClientEvent`, el cual ya hemos definido más arriba.

Ahora que ya se tiene un identificador de aplicación válido, se pueden usar las salas. Para obtener una lista de salas actuales se debe solicitar al servidor que se envíe una lista nueva.

```
void ListarSalas(Connection::id_t id);
```

`ListarSalas` generará un evento de tipo `RoomListEvent`, y este es el contenido:

```
struct RoomListEvent : public Event
{
  Connection::id_t connection;
  std::vector<Room> rooms;
};
```

La estructura `Room` es la siguiente:

```
struct Room
```

```
{
  using id_t = uint32_t;
  using slot_id_t = uint8_t;

  id_t id;
  std::vector<RemoteClient::id_t> slots;
};
```

Existe también la forma siguiente para obtener el listado de salas:

```
future_vector<Room> ObtenSalas(Connection::id_t id);
```

Con el listado, la aplicación puede mostrar al usuario el listado de salas para que este seleccione en cual desea entrar, o si requiere de crear una nueva.

Si un usuario solicita entrar a una sala, la aplicación deberá usar el siguiente método:

```
void UnirseSala(Connection::id_t id, Room::id_t id_sala, Room::slot_id_t
      id_hueco);
```

Como podemos ver, se indica la sala y el hueco, aunque este último es opcional. Si no se indica, se ocupará el primer hueco libre que se encuentre en la sala. Si se consigue entrar en una sala satisfactoriamente, se generará el evento de tipo `JoinedRoomEvent`:

```
struct JoinedRoomEvent : public Event
{
  Connection::id_t connection; //identificador de conexion
  Room::id_t room_id; //identificador de sala
  std::uint8_t slot_id; //indice de hueco ocupado
};
```

En caso de que el usuario decida crear una sala nueva, puede usar el siguiente método:

```
void CreateRoom(Connection::id_t id, Room::slot_id_t num_huecos,
      Room::slot_id_t id_hueco_donde_unirse);
```

De nuevo, si se crea la sala satisfactoriamente, se generará un evento de tipo `JoinedRoomEvent`.

Finalmente, una vez dentro de la sala, un cliente puede comunicarse con el resto de clientes usando los siguientes dos métodos:

```
void EnviarDifusion(Connection::id_t id, const Packet& packet,
      Packet::ReliabilityBitmask mask = Packet::Unreliable);
```

```
void EnviarEntrada(Connection::id_t id, const Packet& packet);
```

Si algún cliente de la sala envía un paquete de difusión, la clase cliente generará un evento de tipo `BroadcastEvent`:

```
struct BroadcastEvent : public Event
{
```

```
  Connection::id_t connection; // identificador de conexion
  milliseconds_t when; // muestra de tiempo de cuando se recibió
  Room::id_t room_id; // identificador de la sala desde donde se manda
  std::uint8_t slot_id; // identificador del hueco desde donde se manda
  Packet data; // los datos enviados por la aplicación
};
```

Nota: `Packet` es una clase auxiliar que almacena un array de bytes. Permite leer o escribir de una forma similar a un iostream de la librería STL de C++, con el añadido de que hace marshalling de los datos. De esta manera damos facilidades a la aplicación usuaria de la librería para evitar preocuparse de endianess u otras cuestiones.

Sin embargo, cuando algún cliente envía una entrada al diario de la sala, la clase cliente generará un evento del tipo `EntriesEvent`:

```
struct EntriesEvent : public Event
{
  Connection::id_t connection; // identificador de la conexión
  Room::id_t room_id; // sala la cual sus entradas han sido actualizadas
  Entry::id_t entry_count; // numero de entradas en la sala
};
```

La forma de obtener una entrada es usando el método siguiente:

```
const Entry& ObtenEntradaDeSala(Connection::id_t id, Entry::id_t entradaId);
```

Esta es la definición de `Entry`:

```
struct Entry
{
  using id_t = Biribit::id_t;

  id_t id; // indice de la entrada dentro del diccionario
  std::uint8_t from_slot; // identificador del hueco desde donde se manda
  Packet data; // los datos enviados por la aplicación
};
```

La aplicación será la encargada de recordar hasta que entrada ha leído con anterioridad, y solicitar una a una las entradas cada vez que el tamaño del diario aumente. Al entrar a una sala nueva, es posible leer las entradas del diario enviadas en toda la existencia de la sala.

PROGRAMA DE PRUEBAS

Una herramienta que es muy útil a la hora de desarrollar una librería es un programa que pruebe cada una de las funcionalidades.

La metodología para el desarrollo de Biribit está basado en las muy usadas metodologías ágiles. Desde el principio del proyecto tenía un listado de las características a programar, ordenadas de tal forma que las primeras satisfacían dependencias de las siguientes. A cada iteración decidía desarrollar una característica. Cada característica se traducía a su parte de código en el servidor, parte de código en el cliente y parte de código en el programa de pruebas. Al final de la iteración,

una vez desarrollada la característica, hacía pruebas exhaustivas para comprobar que tanto la nueva característica como la ya desarrollada funcionasen correctamente.

Aun a pesar de que Biribit no está restringido para juegos, se adecua mucho a ellos por su naturaleza asíncrona y su sistema de eventos. El programa de pruebas fue creado con la idea de que tuviese un comportamiento muy parecido a un juego.

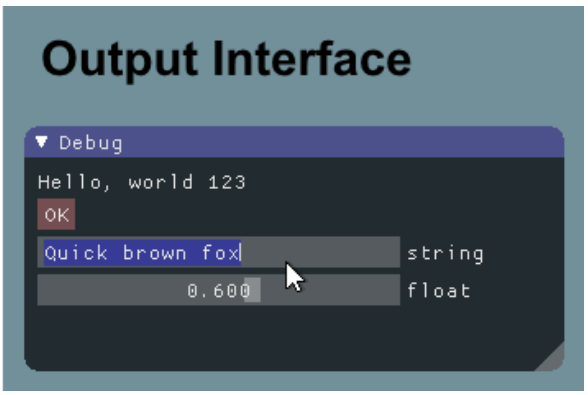
Los juegos son programas que intentan aprovechar el máximo rendimiento de un ordenador. El típico diseño de un juego se define como un programa que itera sin pausa un bucle que procesa la entrada, actualiza el estado y muestra un resultado.

El programa de pruebas está desarrollado con SDL, una librería multiplataforma que permite gestionar ventanas, entradas como teclado, ratón y controladores de juego, y permite pedir al sistema operativo un contexto de OpenGL para aceleración gráfica.

El programa consiste en una serie de interfaces que permiten probar todas las funcionalidades de la librería cliente de Biribit. Para las interfaces utiliza la librería IMGUI, una librería muy ligera que ocupa únicamente un único fichero de código. Las interfaces de IMGUI no fueron pensadas para que fuesen usadas por usuarios, sino que fueron diseñadas desde el principio con la finalidad de probar, depurar y mostrar información adicional sobre el programa. El paradigma de funcionamiento de IMGUI se llama “Modo inmediato”. La ventaja de este método es que el código que describe la interfaz tiene un aspecto muy parecido a la propia interfaz. En la siguiente imagen se muestra un ejemplo:

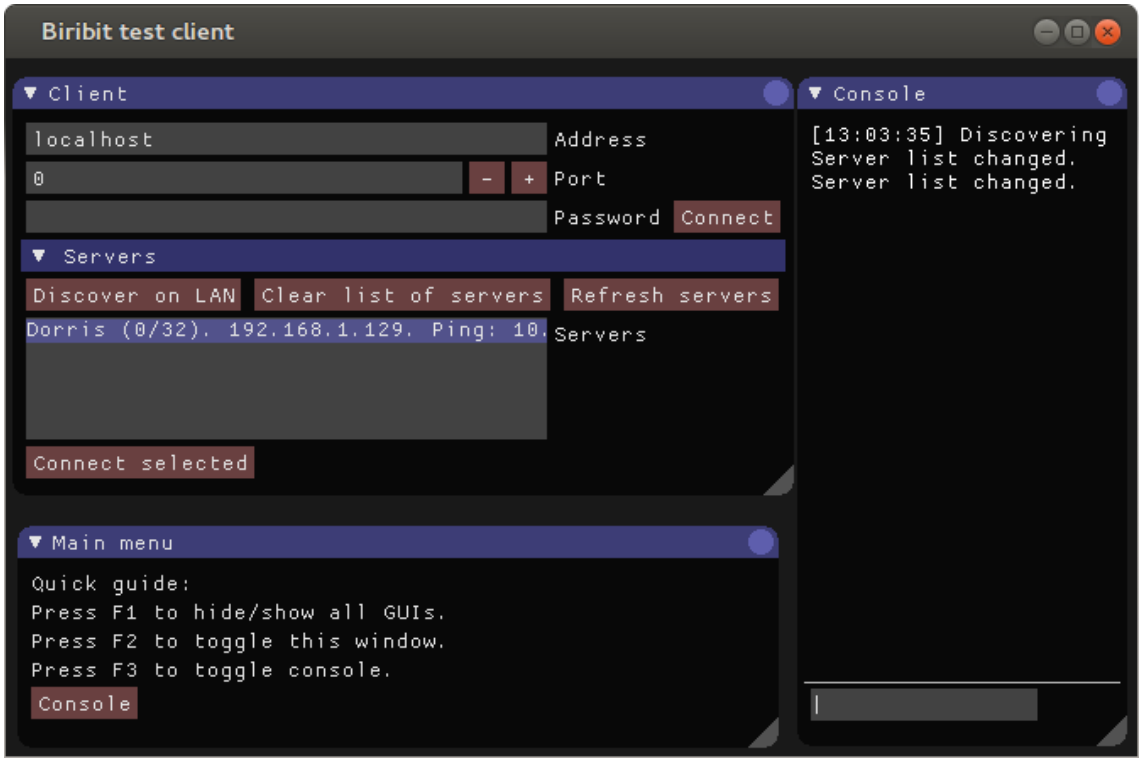
C++ code

```
ImGui::Text("Hello, world %d", 123);

if (ImGui::Button("OK"))
{
    // do stuff
}

ImGui::InputText("string", buf, 256);

ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```



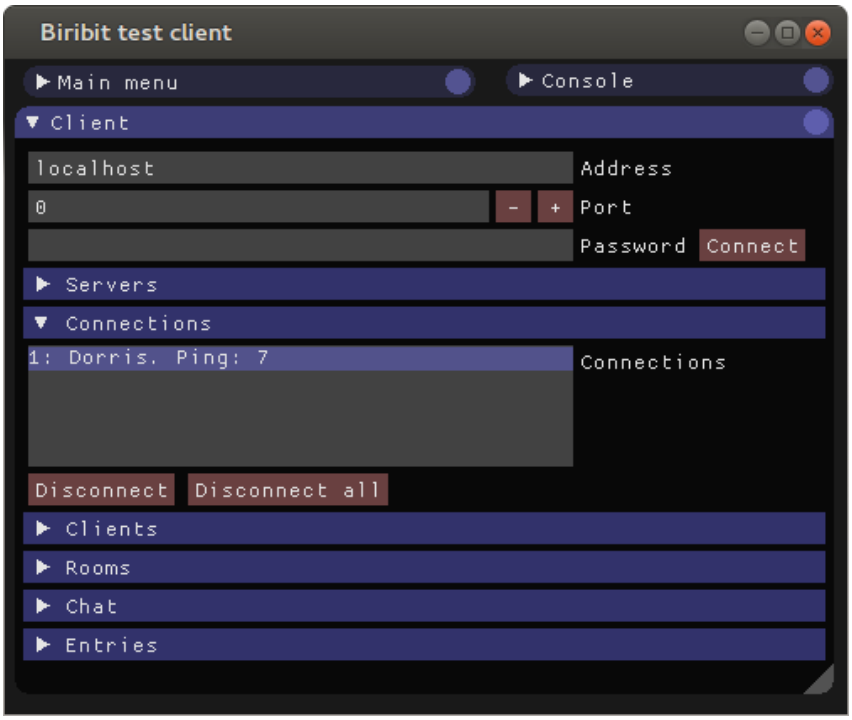
Solamente se requiere que el código de la interfaz se ejecute una vez por iteración del bucle, sin restricción de donde se necesite poner el código.

Volviendo al programa de test, está es la visualización del programa una vez arrancado:

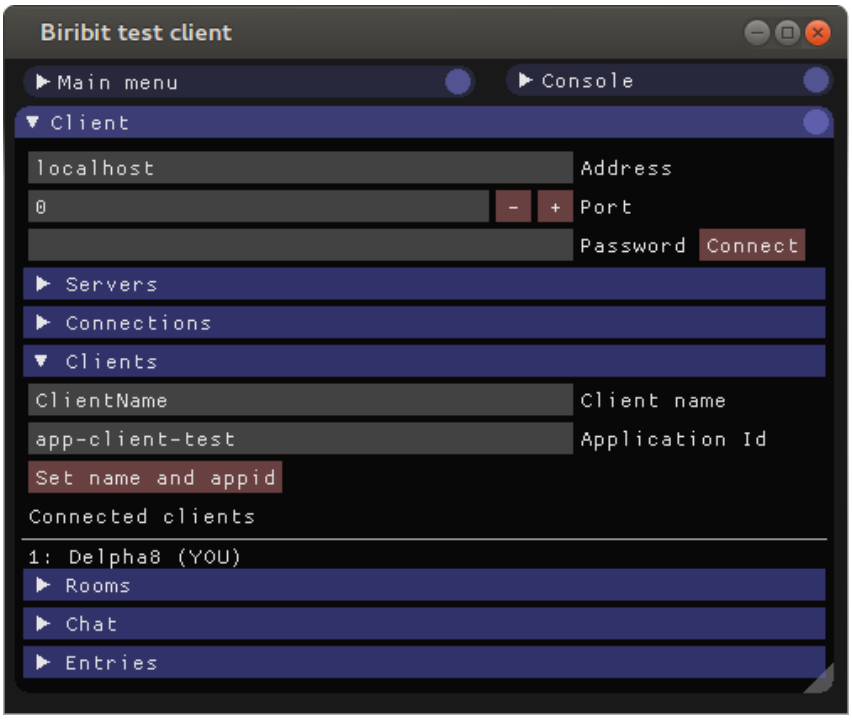


Como se puede ver, el programa tiene una consola para mostrar todos los eventos que el cliente va generando. A su vez, permite ejecutar comandos que se traducen en operaciones sobre el cliente. De todas maneras, todas estas operaciones están plasmadas sobre la interfaz "Client" que se puede ver en la imagen. Esta interfaz, va cambiando en función del estado del cliente y permite acceder a todas las funcionalidades. En la imagen anterior podemos observar cómo se despliegan nuevas secciones como consecuencia de la realización de una conexión. Estas secciones son:

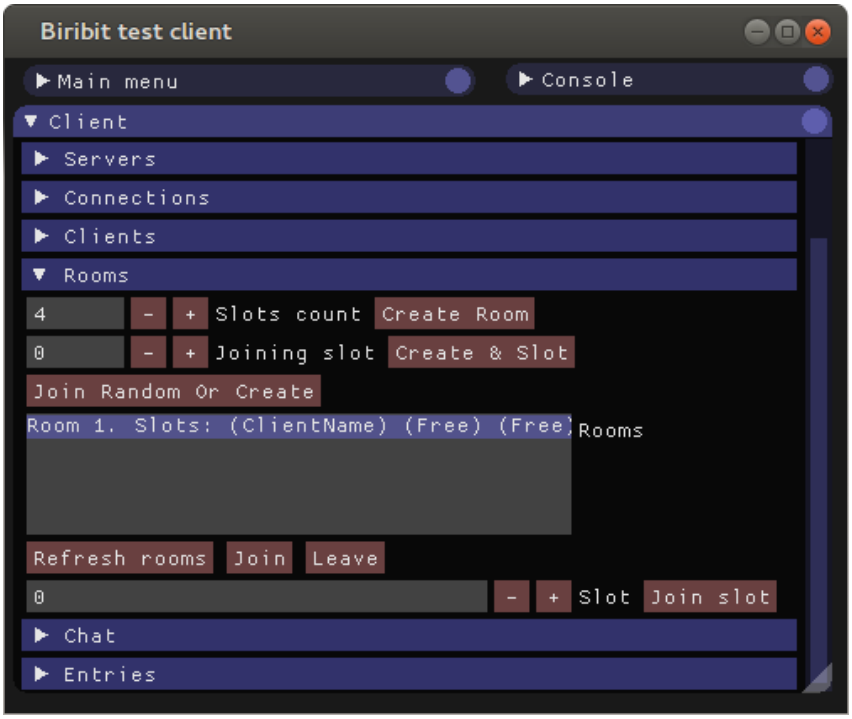
- CONNECTIONS: Se listan las conexiones y permite desconectarse.



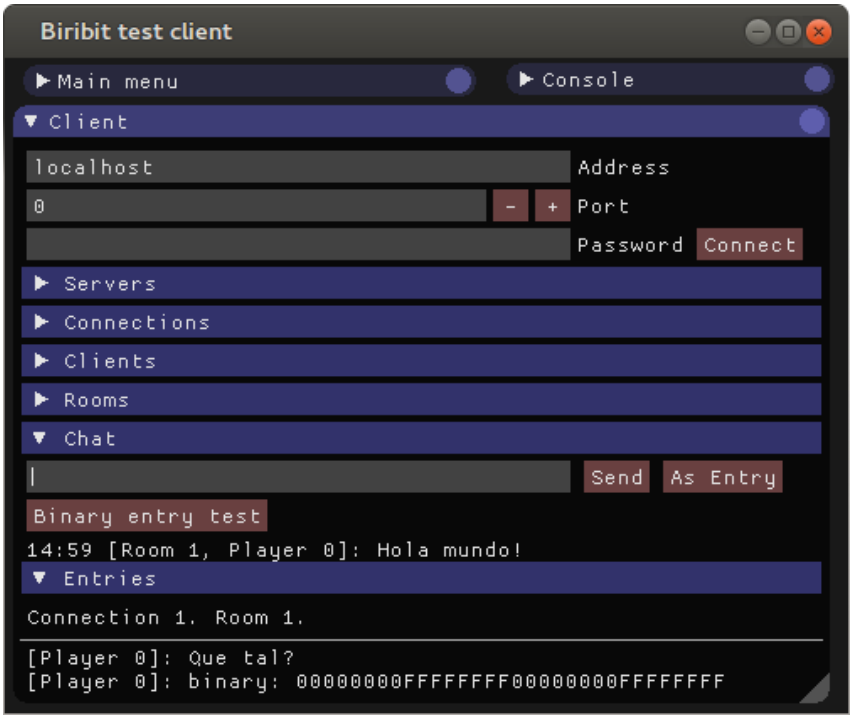
- CLIENTS: Se permite cambiar el nombre y el identificador de la aplicación y listar los clientes. conectados.



- ROOMS: Se permite listar las salas, entrar y crearlas.



- CHAT Y ENTRIES: Se permite a los clientes que están dentro de la sala usar la comunicación por difusión como un chat. Permite a su vez enviar chats como entradas al diario o incluso cadenas binarias. En Entries se muestra las entradas enviadas.



CLIENTE DE BIRIBIT PARA UNITY

Una vez programada la librería cliente, se procedió a la integración del cliente con Unity. Como se ha dicho anteriormente, Unity es un motor para juegos en el que toda la lógica del juego es desarrollada en el lenguaje de programación C#. También recordamos que se forzó a realizar Biribit en C++ para que el servidor fuese lo más eficiente posible y fuese capaz de funcionar en hardware limitado como en una Raspberry Pi. Con esta restricción en mente juntamente con que se decidió usar RakNet como librería de red, forzosamente se tenía que hacer el cliente en C++.

El problema al que nos enfrentamos es el siguiente: Unity permite usar librerías nativas, pero estas, por limitación de C#, deben exportar las funciones en C. Esto no es restrictivo para Biribit, es decir, se permite que la librería esté en C++, pero las funciones exportadas no.

La librería cliente de Biribit exporta las funciones en C++, por lo que se decidió que la mejor opción era crear otra librería totalmente diferente que dependiera de la librería del cliente C++ y que exportara las mismas funcionalidades pero en C. Esta librería se llama BiribitForUnity.

La aproximación para esta librería fue un poco distinta. Para empezar, las clases no existen en C, por lo que, siempre que se quería realizar una operación sobre la librería, se tenía que pasar un puntero llamado “contexto”.

Por ejemplo, si el método Conectar de la clase Cliente requiere de 3 parámetros, la función Conectar de la librería en C requiere de 4. El primer parámetro pasado es siempre el contexto.

Para crear un contexto se añadieron dos nuevas funciones, `CrearCliente` y `DestruirCliente`. Estas no eran más que una capa que realizaba por detrás lo equivalente a un `new` y un `delete`.

Otro cambio importante que se tuvo que hacer es convertir todas las estructuras a C. El mayor problema en este proceso era eliminar todas las apariciones de estructuras `std::string` o

`std::vector`. Las `std::string` se substituyeron por `const char*` y los `std::vector` se sustituyeron por una estructura que tenía un puntero al primer elemento y un entero que indicaba el tamaño.

Pero el cambio más significativo fue el decidir que Unity no tuviese que ir desencolando los eventos de las colas. En su defecto se creó una función `ActualizarEventos` a la que le era pasada una estructura de punteros a funciones. He aquí la estructura:

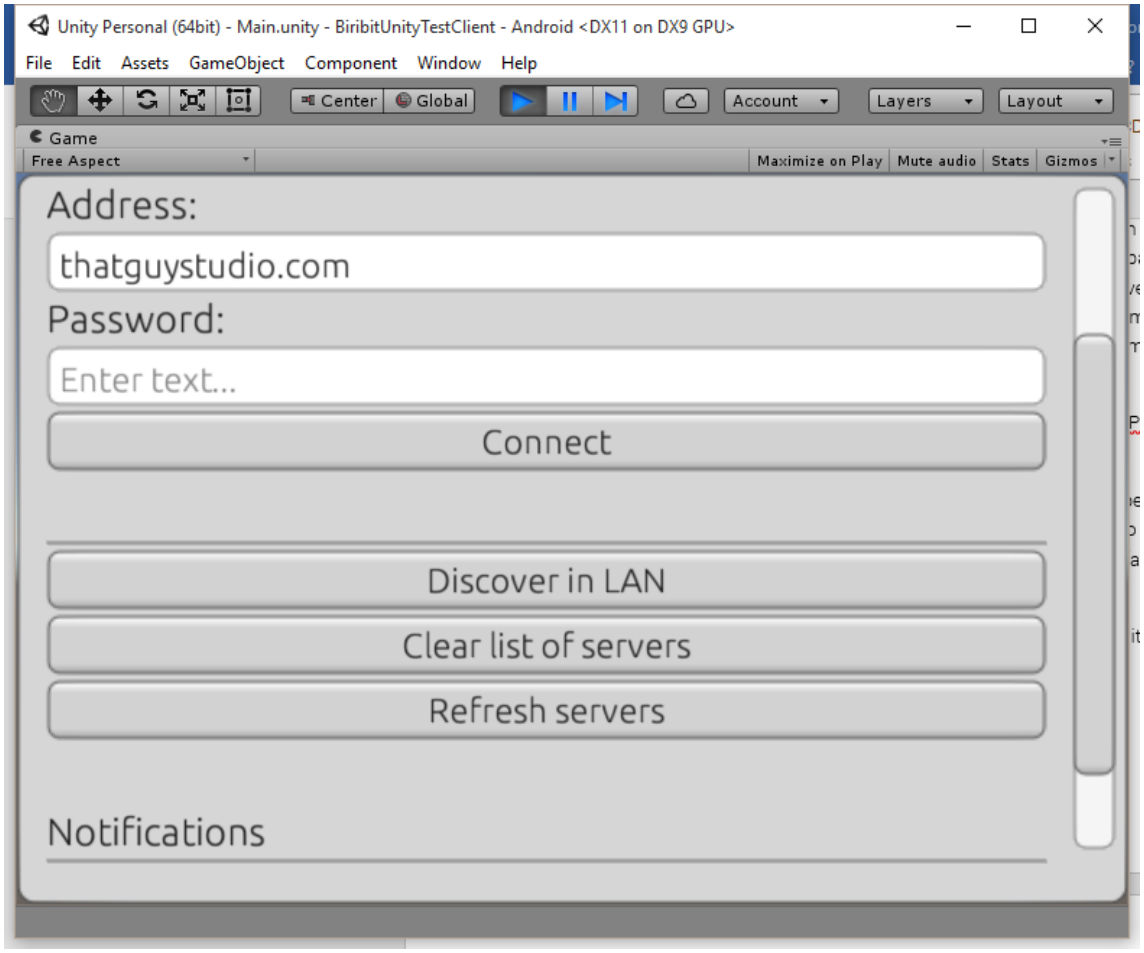
```
struct brbt_EventCallbackTable
{
	brbt_ErrorEvent_callback error;
	brbt_ServerListEvent_callback server_list;
	brbt_ConnectionEvent_callback connection;
	brbt_ServerStatusEvent_callback server_status;
	brbt_RemoteClientEvent_callback remote_client;
	brbt_RoomListEvent_callback room_list;
	brbt_JoinedRoomEvent_callback joined_room;
	brbt_BroadcastEvent_callback broadcast;
	brbt_EntriesEvent_callback entries;
};
```

La función de actualizar leía todos los eventos de una tacada y en función del tipo de evento ejecutaba la función correspondiente. La razón de esta decisión fue por una cuestión de optimización y comodidad:

1. Se evitaba en el código C# de Unity tratar con unions por ser sumamente incómodo, a pesar de que en C la forma más común de hacer que una estructura se pueda interpretar de varias maneras es usando unions.
2. Se evitaba en el código C# seleccionar los casos para cada tipo de evento, ya que cuando una función proporcionada en esta tabla de punteros era llamada, la selección del tipo de evento ya estaba hecha.
3. La llamada a la función C desde C# pasaba como parámetro otras funciones (en C#) encargadas de gestionar cada tipo de evento. Así pues, la función C llamaría, a su vez, a las correspondientes funciones C# para tratar los eventos. De este modo, se evitaba tener que devolver los eventos a la llamada original, cosa que hubiese implicado la necesidad de utilizar memoria dinámica y sus problemas de ineficiencia asociados.

Una vez realizada la librería BiribitForUnity, usando el programa PInvoke Signature Toolkit, se generó un código C# para llamar a las funciones de C. En ese momento ya teníamos control de la librería en Unity, pero no fue suficiente. Sobre el mismo código C# generado por el programa mencionado antes, se hizo una clase Cliente similar a la de C++ que contenía el puntero del contexto y permitía de nuevo llamar a Conectar, pero sin el parámetro del contexto.

Para probar que todas las funcionalidades seguían intactas en Unity, se creó un proyecto similar al programa de pruebas, pero en Unity. En la siguiente imagen se puede ver el aspecto de su interficie.



Unity, como es lógico pensar, necesita que en el proyecto haya la librería de BiribitForUnity compilada para cada plataforma en la que se quiera usar. Estas librerías se deben copiar en el proyecto en cuestión, en Assets/Plugins/Android para Android, o Assets/Plugins/x64 para PC.

En el siguiente apartado hablaremos un poco sobre qué pasos hicieron falta para poder hacer compilaciones fácilmente entre plataformas tan diversas.

CREANDO EL PROYECTO

A sabiendas de que se iba a tener que construir el proyecto para muchos sistemas operativos y con afán de aprender, se decidió utilizar CMake.

CMake [1]es una herramienta similar a Make, pero con la diferencia que está diseñada para ser multiplataforma. Este no construye el proyecto, sino que genera los ficheros necesarios según la plataforma para que se pueda construir. Por ejemplo, en Linux puede generar ficheros Makefile para luego construir con Make, o en Windows puede generar soluciones de Visual Studio para abrirlo con este, editar y generar el binario final.

El motivo de más peso que llevó a usar CMake es que había trabajado con anterioridad con Android. Un problema siempre recurrente en los proyectos en C++ es que nunca acaba siendo el código 100% portable. Muchas librerías incluyen sus propios scripts de construcción, y a veces no

[1] https://cmake.org/

se detienen a mirar si estos funcionan en todos los sistemas operativos. Android en concreto, no es compatible con ningún sistema de construcción de proyectos estándar como Makefile. Este tiene su sistema propio, y además es muy reciente, por lo que hace que ninguna librería de terceros suela estar adaptada para compilar para Android.

El típico problema con el que me había enfrentado más de una vez es tener que ir actualizando los scripts de compilación de Android a la par con el resto. Además, tenía que compilar las librerías de las que dependía el proyecto para cada una de las plataformas y copiar los binarios en el proyecto.

Para evitar esto, se vio que la única alternativa que existía para compilar a través de muchas plataformas, incluyendo Android, era usando CMake. La compilación para Android con CMake no era oficial, sino que se trataba de un proyecto personal de un programador que, mediante un script para Cmake, era capaz de realizar los mismos pasos que los scripts de Android. Esto añadió dificultad al proyecto porque estaba pobremente documentado y requería configurar muy bien el entorno de variables globales de Windows/Linux.

A su vez, se ha intentado evitar compilar RakNet o Protobuffers manualmente para todas las plataformas y configuraciones posibles (x86, x64, ARM, Debug, Release, etc). Se han incluido estos directamente en el proyecto de CMake haciendo manualmente y de cero los scripts de compilación. Estos han supuesto un reto bastante grande, sobre todo en el caso de Protobuffers. Ha llevado 2 semanas pulir los scripts de CMake, pero finalmente, una vez funcionando a la perfección, ha agilizado todo el proceso de desarrollo.

Para el control de versiones de Biribit se ha usado Git[2]. Este es un control de versiones que nació juntamente con el desarrollo del kernel de Linux. Se ha popularizado últimamente porque no requiere de servidor centralizado.

Las librerías RakNet[3] y Protobuffers[4] tienen el código público y de libre acceso mediante repositorios Git en Github. Esto ha sido sumamente útil porque ha permitido enlazar ambos repositorios como submódulos y ha ahorrado incluir el código en el repositorio. Además, esto permite actualizar ambas librerías fácilmente en futuras versiones del proyecto.

Los scripts de CMake se componen de los siguientes proyectos:

- RakNet: La librería de red, la cual se enlaza como una librería estática en el cliente y en el servidor.
- BiribitCommon: Código en común entre el cliente y el servidor, que incluye la clase Packet, utilidades para hilos de ejecución, hasta los mensajes del protocolo.
- ProtoMessages: Este proyecto incluye tanto el código de Protobuffers como el código C++ generado a partir de todas las estructuras proto.
- BiribitServer: El ejecutable del servidor.
- BiribitForUnity: Una capa de abstracción de la librería cliente, pero en C.
- BiribitClient: La librería cliente.

[2] https://git-scm.com/
[3] https://github.com/OculusVR/RakNet
[4] https://github.com/google/protobuf

- ClientTest: El programa de pruebas.

La mayoría del desarrollo del juego se realizó desde Windows usando Visual Studio 2015. La razón es porque la versión de Unity que se usó solamente tenía editor para Windows. Si era necesario hacer pruebas que implicaran compilar el proyecto y probar en Unity, no quedaba otra opción que programar en Windows.

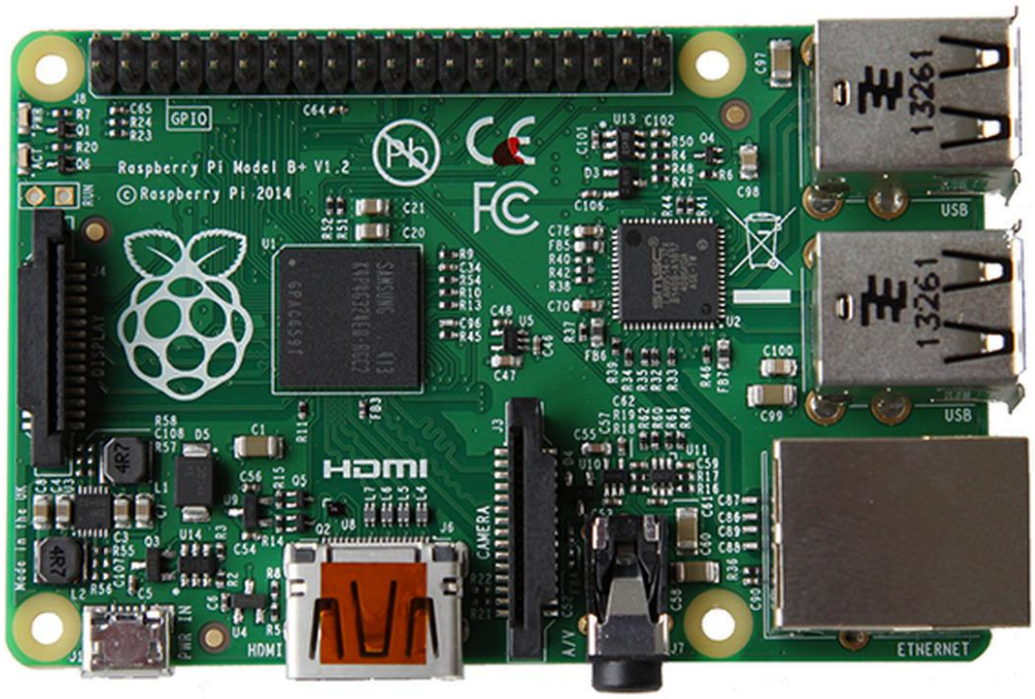
CONFIGURANDO LA RASPBERRY PI

A la par que se ha ido desarrollando Biribit, fui informándome y mirando qué tenía que comprar para el proyecto. Al parecer no todos los adaptadores Wi-Fi USB podían funcionar así que hubo un tiempo previo de investigación al respecto.

Aun no estando acabado Biribit, tuve ocasión de ir probando el servidor en la Raspberry Pi, pero no fue hasta el final de este que rematé los detalles creando un servicio de Biribit y permitiendo que nada más conectando la Raspberry a la electricidad, todo arrancase de forma automática sin supervisión.

CONFIGURANDO EL PUNTO DE ACCESO

Los componentes usados para esta configuración fueron: Raspberry Pi 2, Dongle Wi-Fi USB, MicroSD 4GB y un PowerBank de 22.000 mAh.



Visitando la página web de la Raspberry pude descargar la imagen de 512MB del sistema operativo Raspbian, un GNU/Linux Debian adaptado para la Raspberry Pi. La instalación fue fácil, descargabas la imagen y hacías un volcado de la imagen en la tarjeta MicroSD. A continuación, conecté la Raspberry a una pantalla de ordenador, la conecté a la red con un cable Ethernet, le enchufé un teclado, le introduje la MicroSD y la conecté a la corriente con un simple cargador de móvil MicroUSB. Poco después, arrancó permitiendo configurar el usuario y la contraseña, y ampliar la partición de la MicroSD para aprovechar todo el espacio.

Una vez arrancada me dispuse a configurarla como punto de acceso Wi-Fi. Buscando por internet encontré las instrucciones sobre cómo configurarla como punto de acceso en Adafruit[5], una página web americana que se ocupa de vender y distribuir Raspberrys.

Durante el tutorial, primero explica como instalar dhcpd, un servidor DHCP, configurarlo de tal manera que proporcione por el dispositivo wlan0 desde la IP 192.168.42.10 hasta la IP 192.168.42.50. Para los DNS proporcionamos los DNS públicos de Google (8.8.8.8 y 8.8.4.4).

Más tarde configuré el dispositivo wlan0 para que tenga una IP estática 192.168.42.1. Finalmente procedí a instalar Hostapd, un servicio para Linux que permite crear servidores de autentificación Wi-Fi y puntos de acceso. El nombre de la Wi-Fi configurada (ESSID) fue "DivineWifi" y la contraseña fue "DivinePass".

Después de eso, configuré las IP tables para crear un NAT entre los dispositivos conectados desde la red Wi-Fi y la conexión de cable Ethernet, de tal manera que si la Raspberry tenía internet por el cable, proporcionaba internet a los dispositivos conectados por Wi-Fi.

Todos estos servicios fueron configurados para arrancar cuando el sistema arrancase. Finalmente acabé el tutorial y procedí a probar.

Tras realizar todos los pasos, no funcionó. El dispositivo wlan0 no era capaz de arrancar en el modo en el que hostapd requería para proporcionar Wi-Fi. Sin rendirme, busque información sobre el chip del dongle Wi-Fi y leí en uno de los comentarios del mismo tutorial que debía compilar mi propia versión de hostapd para que funcionase.

No fue difícil, bajé el código de una página oficial del fabricante, ejecuté las instrucciones indicadas, instalé el nuevo binario de hostapd que generó y funcionó.

HACIENDO DEL SERVIDOR DE BIRIBIT UN SERVICIO

Después de eso, simplemente descargué el código de Biribit y lo compilé para la Raspberry Pi. He de destacar que me sorprendió la diferencia de tiempo de compilación entre el ordenador y la Raspberry. Esta tardo aproximadamente 1h en compilar RakNet y Protobuff, pero tardó pocos minutos en compilar el servidor de Biribit.

Una vez generado el binario del servidor de Biribit, copié este en /usr/bin como si un binario del sistema se tratase, y copié uno de los scripts de arranque de /etc/init.d llamándolo Biribit.

Este es el resultado:

```
#!/bin/sh

### BEGIN INIT INFO
# Provides:             biribit
# Required-Start:       $remote_fs
# Required-Stop:        $remote_fs
# Should-Start:         $network
# Should-Stop:
# Default-Start:        2 3 4 5
```

[5] https://learn.adafruit.com/setting-up-a-raspberry-pi-as-a-wifi-access-point/install-software

```
# Default-Stop:         0 1 6
# Short-Description:    Biribit Server
# Description:          Biribit Generic Game Server
#                       Authenticator
### END INIT INFO

PATH=/sbin:/bin:/usr/sbin:/usr/bin
DAEMON_SBIN=/usr/bin/biribit
#DAEMON_DEFS=/etc/default/biribit
DAEMON_CONF="-n DivineAndConquer"
NAME=biribit
DESC="Biribit Server"
PIDFILE=/var/run/biribit.pid

[ -x "$DAEMON_SBIN" ] || exit 0
#[ -s "$DAEMON_DEFS" ] && . /etc/default/hostapd
#[ -n "$DAEMON_CONF" ] || exit 0

DAEMON_OPTS="-i $PIDFILE $DAEMON_CONF"

. /lib/lsb/init-functions

case "$1" in
  start)
        log_daemon_msg "Starting $DESC" "$NAME"
        start-stop-daemon --start --oknodo --quiet --exec "$DAEMON_SBIN" \
                --pidfile "$PIDFILE" -- $DAEMON_OPTS >/dev/null
        log_end_msg "$?"
        ;;
  stop)
        log_daemon_msg "Stopping $DESC" "$NAME"
        start-stop-daemon --stop --oknodo --quiet --exec "$DAEMON_SBIN" \
                --pidfile "$PIDFILE"
        log_end_msg "$?"
        ;;
  reload)
        log_daemon_msg "Reloading $DESC" "$NAME"
        start-stop-daemon --stop --signal HUP --exec "$DAEMON_SBIN" \
                --pidfile "$PIDFILE"
        log_end_msg "$?"
        ;;
  restart|force-reload)
        $0 stop
        sleep 8
        $0 start
        ;;
  status)
        status_of_proc "$DAEMON_SBIN" "$NAME"
        exit $?
        ;;
  *)
        N=/etc/init.d/$NAME
        echo "Usage: $N {start|stop|restart|force-reload|reload|status}" >&2
        exit 1
        ;;
esac

exit 0
```

El script funciona de la siguiente manera:

Las cabeceras del principio son el estándar Init Script LSB[6]. Indica el nombre servicio que provee y pide que servicios requiere antes de arrancar. En nuestro caso pedimos que la red esté funcionando.

El script ejecuta lo que hay en la variable $DAEMON_SBIN con los parámetros que hay en la variable $DAEMON_OPTS. En nuestro caso es tal que:

```
/usr/bin/biribit -i /var/run/biribit.pid -n DivineAndConquer
```

El servidor de Biribit tiene dos formas de funcionar: modo consola y modo servicio. Cuando se le proporciona un fichero con el parámetro –i, este funciona en modo servicio. Este modo solo existe en Linux. Para que un proceso pueda funcionar como servicio tiene que crear un proceso clon (usando fork) y desanclarse del padre. El proceso hijo escribirá su PID en el fichero proporcionado y se desanclará del padre cerrando todos los canales de entrada estándar, salida estándar y error estándar. Cuando el hijo se desancle, el proceso padre se cerrará, devolviendo el foco al terminal o proceso que ha ejecutado el programa.

Como hemos indicado, en modo servicio no hay salida estándar, pero como hemos comentado en el apartado de Biribit, podemos fácilmente redirigir toda la salida del servidor a un fichero de texto que usaremos como registro.

El script creado nos permite arrancar el servidor, pararlo, o reiniciarlo. Y gracias a que existe el fichero PID, será capaz de saber si el servidor está arrancado y como pararlo a partir del PID.

Una vez comprobado que el script funciona, mediante un simple comando se añadieron links simbólicos en todos los runlevels para que el servidor arrancase nada más encenderse la Raspberry Pi.

Como futuro trabajo ha quedado realizar un servidor HTTP para que cualquier persona que se conecte a la red pueda visitar y descargarse la última versión del juego. Esta tarea no es realmente muy laboriosa, pero se ha prescindido de hacerla dado que el juego estaba en pleno desarrollo y aun no existía ninguna versión que pudiese poner disponible mediante el servidor HTTP.

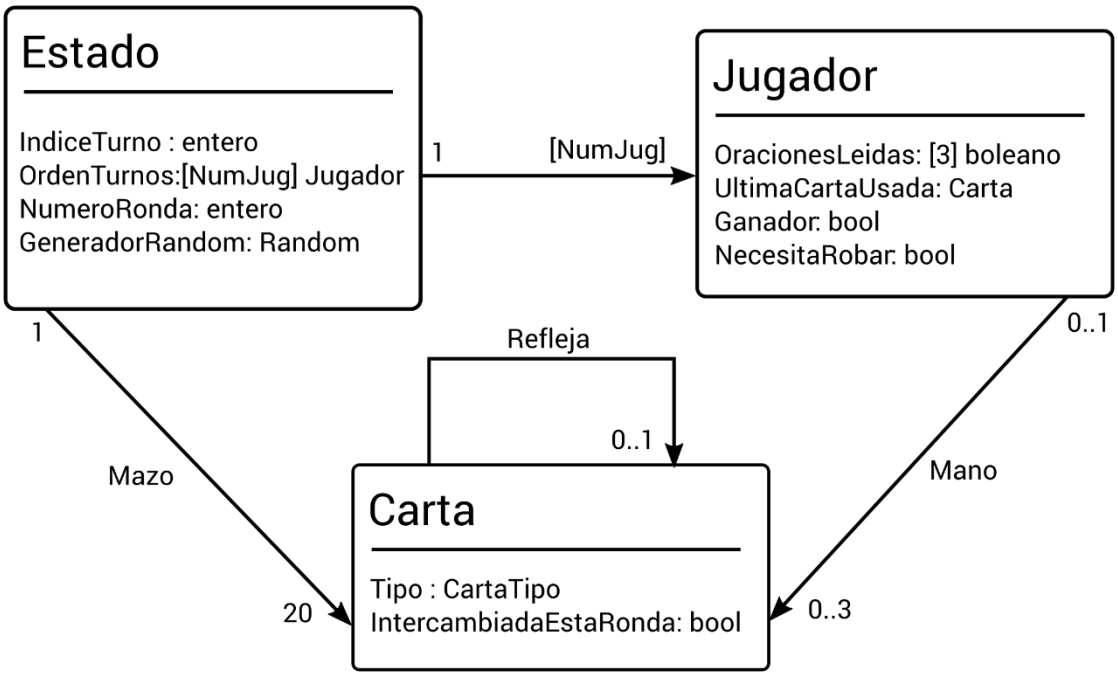
[6] https://wiki.debian.org/LSBInitScripts

DESARROLLO DEL JUEGO EN UNITY3D

Finalmente, tras mucho tiempo desarrollando Biribit, se empezó el desarrollo del juego. Las piezas del puzzle estaban creadas así que hemos de reconocer que los apartados que vienen a continuación se han desarrollado bastante más rápido en comparación a Biribit.

ESTADO DEL JUEGO

Para empezar, se ha creado un proyecto aparte que permitiera programar el estado del juego e ir probando que este se iba comportando según se había marcado en las normas. Con el diseño de clases siguientes, se ha recreado el estado del juego:



La primera consideración es que el juego debía de ser determinista, es decir, si se realizaban unas operaciones determinadas con los mismos parámetros en diferentes ejecuciones, el estado final tenía que ser el mismo. No es difícil cumplir esto, pero hay que tener cuidado porque desde el primer momento que se incluyen eventos aleatorios, se tiene que forzar que la secuencia de aleatorios sea la misma para toda la partida. Se incluyó un generador de números aleatorios y este era inicializado con una semilla al principio de la partida.

Las funciones en C# que podían realizar cambios sobre el estado son las siguientes, las llamaremos operaciones:

```
void EmpezarJuego(int semilla, int numJugadores);
void RobarCarta(int indiceJugador, int indiceCarta);
void UsarCarta(int indiceJugador, int indiceCarta, params int[] extra);
void IntercambiarCarta(int indiceJugador, int indiceCartaJugador, int
      indiceCartaObjetivo);
```

Y las funciones para consultar el estado son las siguientes, las llamaremos consultoras:

```
public bool ObtenAlgunJugadorNecesitaRobar()
public bool[] ObtenJugadoresQueNecesitanRobar()
public int ObtenIndiceJugadorGanador()
public int DeQuienEsElTurno()
public VistaCarta ObtenCarta(int indiceCarta);
public VistaJugador ObtenJugador(int indiceJugador);
```

La operación de empezar juego, como se ha destacado antes, fuerza a empezar el generador de números aleatorios con una secuencia a partir de la semilla, de tal manera que todos los jugadores en su estado puedan recrear la misma secuencia de números aleatorios.

Respecto al resto de operaciones, aun a pesar de que el estado del juego conoce de quien es el turno, se ha preferido que se deba indicar que índice de jugador hace la acción por el siguiente motivo. Cuando un jugador realiza una acción desde la interfaz del juego, esta se envía al servidor. Hasta que el jugador no recibe de vuelta la acción del servidor, el estado del juego local no se actualiza y, por lo tanto, tampoco la interfaz. Recibir tu acción de vuelta desde el servidor es una confirmación de que todos los otros jugadores lo han recibido. Si un jugador es muy rápido volviendo a marcar una acción, puede realizar varias acciones antes de que el estado del juego local se actualice. Si esto pasase, se enviarían dos acciones, las dos las aceptaría el servidor, pero el estado del juego ignorará la segunda acción porque comprobará que el índice del jugador que hace la segunda acción no es el índice del jugador que debería hacer el turno en ese momento.

El estado del juego no deja realizar ninguna acción como `UsarCarta` o `RobarCarta` hasta que `ObtenAlgunJugadorNecesitaRobar` devuelva falso. Para ello, todos los jugadores que no tengan las 3 cartas podrán usar `RobarCarta` aunque no sea su turno hasta que todos tengan las 3 cartas. Si un jugador justamente acaba de hacer una acción, perderá una carta, y de nuevo el siguiente jugador no podrá hacer la acción hasta que todos los jugadores no tengan 3 cartas.

El juego debería acabar cuando el estado del juego indica con `ObtenIndiceJugadorGanador` que hay un ganador. El índice -1 indica “Ningún Jugador”.

Como se puede observar, las operaciones no devuelven ningún booleano notificando si la acción ha sido satisfactoria o no. Se ha optado por usar un método distinto de notificar las acciones efectivas realizadas en el juego usando un patrón observador.

```
public interface Observador
{
  void CartaUsada(int indiceJugador, int indiceCarta);
  void CartaIntercambiada(int indiceJugador, int indiceCartaOrigen, int
      indiceCartaObjetivo);
  void CartaRobada(int indiceJugador, int indiceCarta);
  void Radar(int indiceJugador);
  void RevelarCarta(int indiceJugador, int indiceCarta, TipoCarta carta);
  void FinDeRonda(TipoFinDeRonda tipo);
}
```

Cualquier clase que heredase de esta interfaz de Observador podía inscribirse a los eventos del estado del juego. Esta manera era más cómoda porque si se producía más de un suceso en una acción, era más fácil notificar estos.

Finalmente, como última herramienta de depuración, se han definido todas las precondiciones de todas las operaciones del juego, y se ha forzado el estado del juego para generar excepciones en caso de que estas precondiciones no se estuviesen cumpliendo. Gracias a esto, junto con el modo depuración, tras varios juegos de prueba se ha conseguido eliminar todos los errores del código, cosa que nos ha permitido proceder al siguiente paso.

SERIALIZACIÓN DE LAS OPERACIONES

Una vez programado el estado del juego, ha sido necesario programar una capa que codificase las operaciones del estado de juego como `EmpezarJuego`, `RobarCarta`, `UsarCarta` o `IntercambiarCarta` en una cadena de bytes que pudiese ser enviada por red y reinterpretada para realizar la acción correspondiente.

Esta parte ha sido fácil, puesto que el juego se podía simplificar en 4 acciones, y se optó por codificar a mano las operaciones con un identificador 0, 1, 2 o 3 en función de que operación era, seguido de sus parámetros. Para realizar esta tarea se ha creado la siguiente clase:

```
public class Serializador
{
  static public byte[] EmpezarPartida(int semilla, int numJugadores);
  static public byte[] UsarCarta(int indiceCarta, params int[] extraParms);
  static public byte[] RobarCarta(int indiceCarta);
  static public byte[] IntercambiarCarta(int indiceCarta1, int indiceCarta2);
  static public void Deserializar(byte[] serializacion, EstadoJuego estado);
}
```

Esta cadena de bytes era enviada como entrada al diario de la sala, y una vez se confirmaba que había una entrada nueva, se leía del servidor y se mandaba deserializar. De esta manera no se produciría una acción sin confirmación del servidor.

CREACIÓN DEL JUEGO

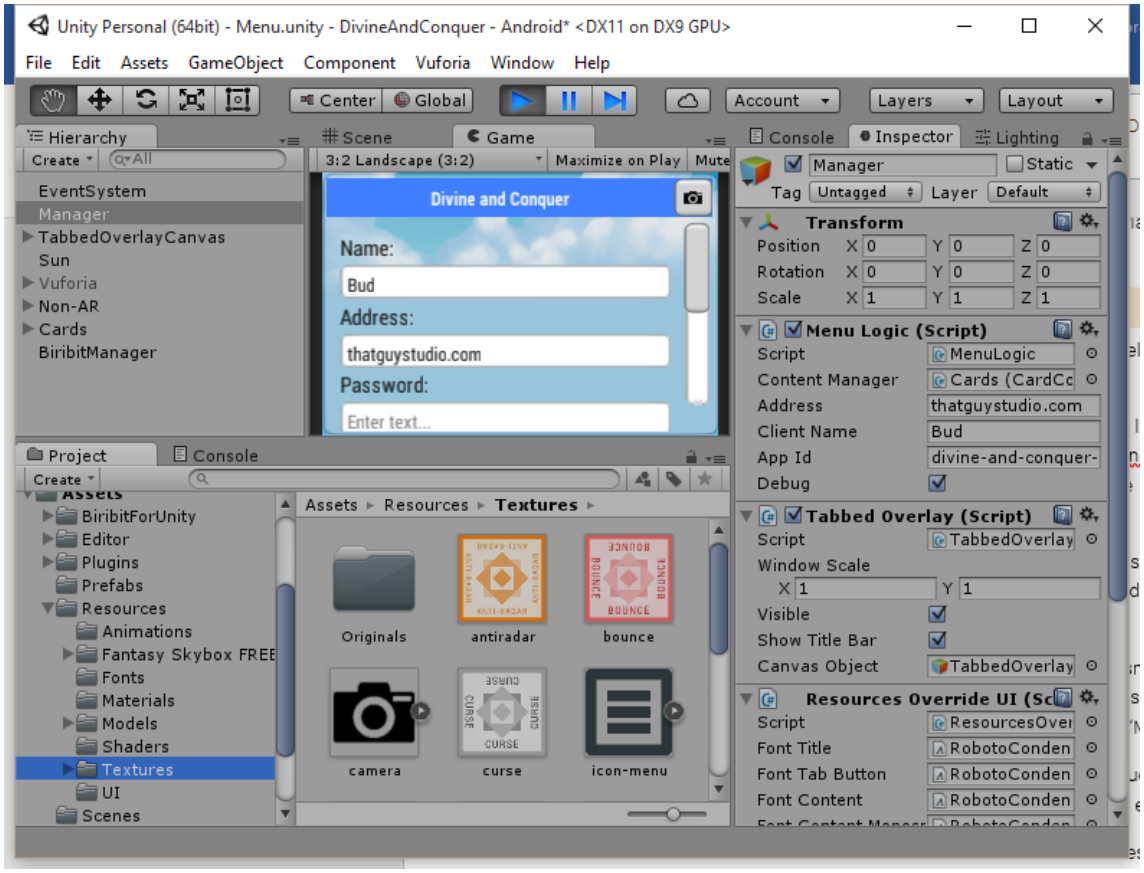
Una vez hechas estas dos partes, ahora únicamente faltaba crear el proyecto de Unity3D, juntar todas las partes y comprobar su funcionamiento.

El juego de Unity es el que une toda la parte de Biribit junto con la parte de la lógica del juego ya programada en el apartado anterior. El proyecto se llama DivineAndConquer y consiste en una interfaz que el usuario puede utilizar para conectarse fácilmente al servidor, crear partidas y empezarlas.

Para dibujar la interfaz se han utilizado los módulos de interfaz ya existentes de Unity, llamados Unity UI. Estos son del tipo “Modo persistente” a diferencia del “Modo inmediato” del que hablamos en el apartado del programa de pruebas.

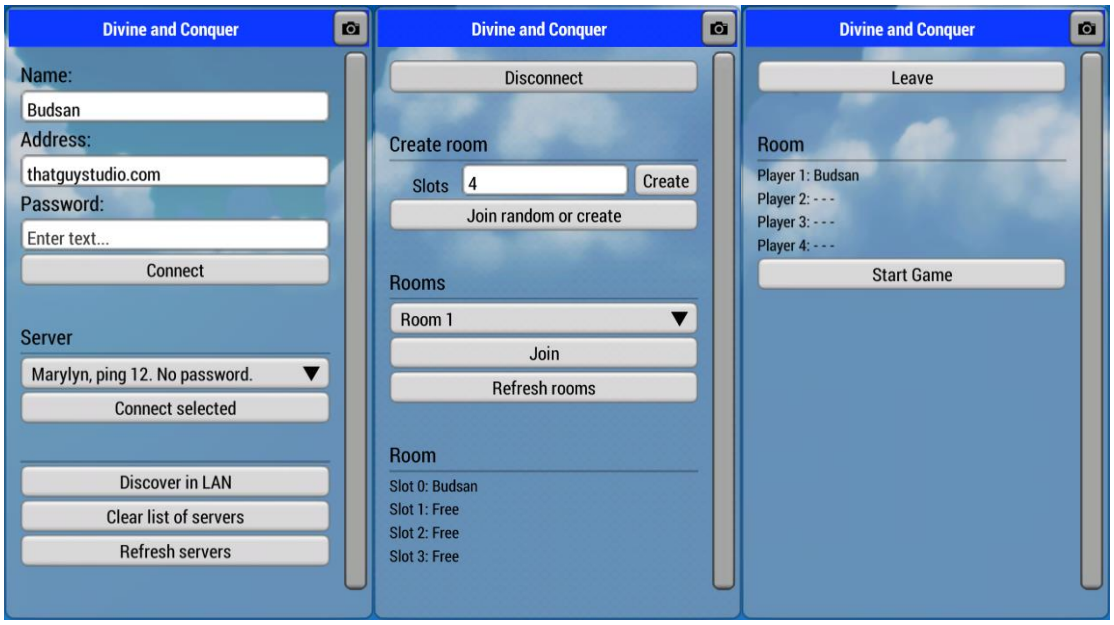
Todas las interfaces en Unity son creables y editables desde el mismo editor, y no desde código. Sin embargo, y por comodidad se ha optado por usar SilverUI, una librería desarrollada por mí en el pasado que permite trabajar con las interfaces de Unity como si fuesen en “Modo inmediato”.

SilverUI permite crear ventanas, las cuales, de la misma forma que vimos con IMGUI, permiten añadir elementos de interfaz, tales como campos de texto, botones, etcétera.



En la imagen de arriba podemos ver el juego arrancando desde el editor de Unity.

La interfaz muestra contextualmente información en función del estado del juego. Tiene cuatro interfaces diferenciadas:



- MENÚ DE CONEXIÓN: Permite introducir el nombre que se desea mostrar en partida, la dirección IP o nombre de dominio donde conectarse. También permite encontrar servidores en red local y conectarse a ellos.

- MENÚ DE SERVIDOR: Una vez realizada una conexión, este menú permite listar salas, unirse o crear nuevas.
- MENÚ DE SALA: Cuando el jugador se ha unido a una sala, se muestra este menú. Este es el más simple de todos, muestra los nombres de los jugadores que se han unido y permite empezar la partida con un botón. Una vez empezada la partida, se creará una partida para tantos jugadores como personas hay en la sala.
- MENÚ DE JUEGO: Cuando el jugador está unido a una sala y el juego esta empezado, se muestra este menú, que contiene toda la información que el juego permite ver al jugador.
    - En la parte de arriba hay un listado con todas las notificaciones que han ido ocurriendo en el juego como, quien ha hecho el último turno y que carta usó, cuando acaban las rondas o que jugador ha generado un ruido de radar.
    - Más abajo muestra la mano actual del jugador. Si no se tienen cartas, se sustituyen los huecos donde van las cartas por "vacío".
    - A continuación, se muestran las oraciones ya leídas. Hay 3 cajitas, cada una indica la primera, segunda y última oración respectivamente. Cuando una cajita está marcada significa que esa oración ha sido leída.
    - La última parte de este menú es la parte de las acciones. Esta parte es totalmente contextual y tiene varios estados:
        - Si el jugador tiene algún hueco "vacío" en la mano, muestra un teclado numérico donde permite robar las cartas que no están en juego.
        - Si algún jugador tiene algún hueco "vacío", se muestra que otros jugadores necesitan robar.
        - Si todos los jugadores tienen 3 cartas, pero no es el turno del jugador, se muestra de quien es el turno.
        - Si todos los jugadores tienen 3 cartas y es el turno del jugador salen 4 botones: Las 3 cartas más un botón de "Intercambiar".

Si se pulsa sobre el botón "Intercambiar" este quedará marcado y se mostrará un menú abajo para seleccionar una de las cartas del jugador. Cuando se marca una carta del jugador, se mostrará para todos los jugadores 3 botones con los números de sus cartas. Seleccionando uno de estos realizara el intercambio entre la carta del jugador seleccionada y otra carta seleccionada de otro jugador.

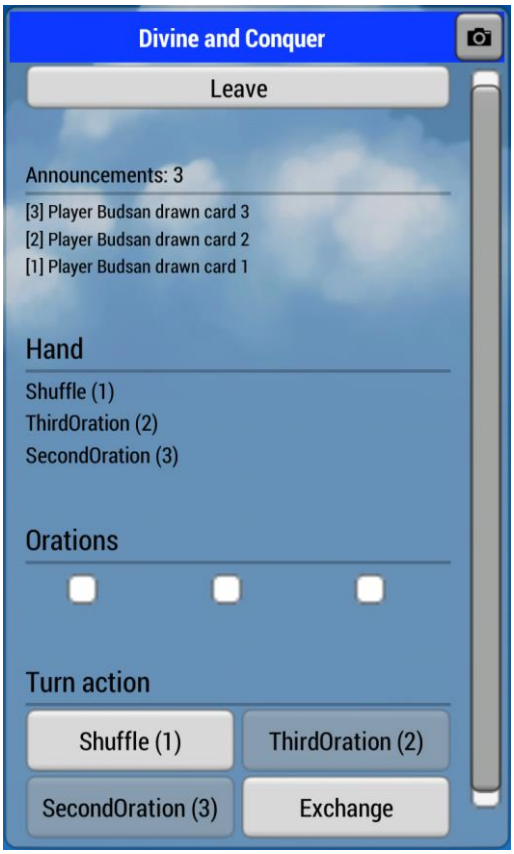
Apretando sobre un botón de carta, usará la carta. Algunas cartas requieren de más información, por lo que se mostrará el botón como seleccionado y aparecerán abajo más opciones. Estas opciones dependen de la carta. Por ejemplo: La carta "Revelar" mostrará por cada jugador 3 botones con los números de sus cartas. Seleccionando uno de estos botones, el jugador indicará al juego que se quiere usar la carta "Revelar" sobre la carta indicada.

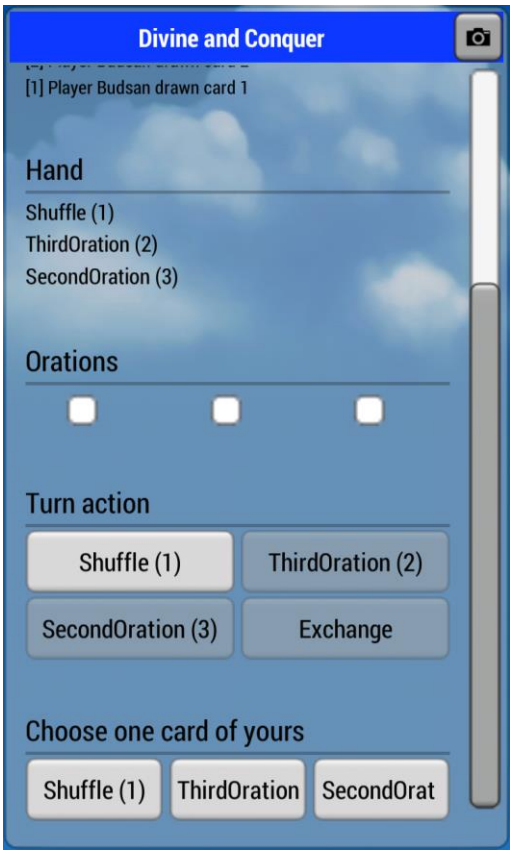
Tan pronto como un jugador gane, la partida acabará, mostrando el menú de sala e indicando que jugador ha ganado.

La interfaz está programada en una única clase, tiene un cliente Biribit y esta misma se ocupa de capturar los eventos. Por otro lado, también usa el serializador y el estado del juego mencionado en el apartado anterior.

Gracias a que ambas partes están probadas y son funcionales, el desarrollo del juego simplemente se ha limitado a crear la interfaz que unía las partes, y a partir de ahí el proyecto entero cogió forma.

Para acabar, adjunto dos imágenes sobre la interfaz del menú del juego.

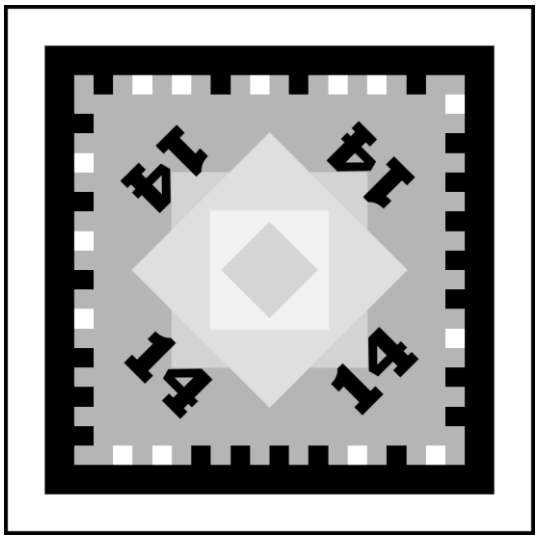




INTEGRACIÓN DE VUFORIA

El último paso a realizar en el proyecto era la realidad aumentada. Como ya se indicó en el principio del documento, se ha utilizado Vuforia como herramienta para la realidad aumentada. Esta es una librería hecha en C++ por Qualcomm, pero incluye un paquete de integración para Unity. Vuforia tiene 2 métodos para detectar elementos mediante la cámara: Frame markers y Image Targets.

FRAME MARKERS son unas imágenes predefinidas que Vuforia puede reconocer y seguir. Un ejemplo sería la mostrada a la izquierda. Esta es una imagen cuadrada con un marco negro donde en la parte interior, tocando con el marco negro hay unos pequeños cuadrados blancos o negros. Estos cuadrados blancos y negros indican el identificador, y soporta hasta 512 identificadores. La imagen que hay central es irrelevante siempre que no dificulte la detección de los cuadrados blancos y negros, por lo que se recomienda que haya un alto contraste de color. Este tipo de imagenes son las usadas para las cartas del juego.



El juego tiene 20 cartas, del 1 al 20, que corresponden frame markers del 0 al 19. El diseño utilizado para las cartas se puede ver en la imagen de la derecha. Este diseño se reutilizó para mostrar las cartas virtuales, quedando todo el diseño homogéneo entre las cartas físicas y las cartas virtuales.

IMAGE TARGETS son imágenes arbitrarias que Vuforia puede reconocer y seguir. Al contrario que los frame markers o cualquiera de los markers tradicionales que siempre se han usado en las librerías de realidad aumentada, estas no requieren de zonas claramente contrastadas en blanco o negro para que se puedan reconocer. Vuforia detecta las características que se encuentran de forma natural en la imagen y son reconocidas si estas características están dentro de una base de datos construida previamente.

Usualmente se utiliza para reconocer carteles publicitarios de los cuales se desea que desde una aplicación móvil se vea un elemento virtual que añada contenido adicional al anuncio o muestre promociones. En el caso de este juego se deseaba usar para los mini juegos en tiempo real.

La idea era que, durante el transcurso de una partida, hubiese un tapete en medio de la mesa de un tamaño de 50x50cm con una imagen arbitraria que el juego fuese capaz de reconocer. Cuando se activará un mini juego, todos los jugadores deberían apuntar al tapete para indicar que están preparados para jugar y cuando todos estuviesen preparados, aparecería un mini juego virtual en este tapete de unos 10 segundos donde los jugadores deberían puntuar lo más alto posible. Por

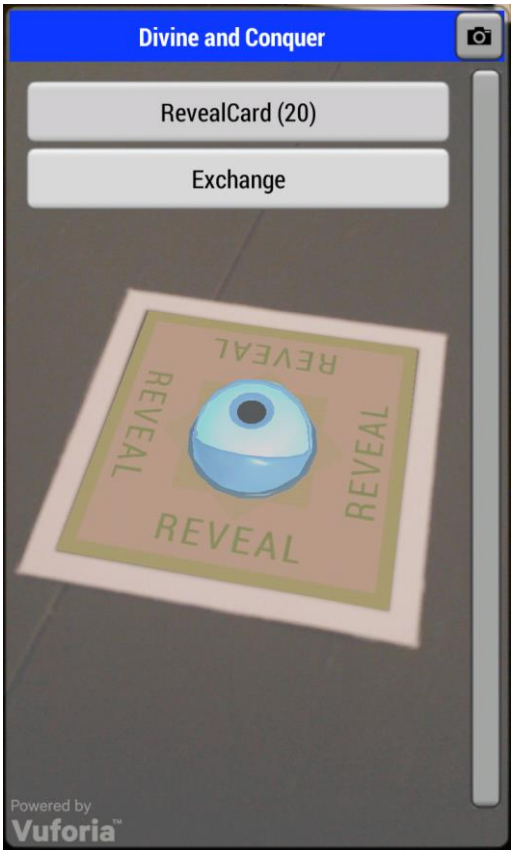
falta de tiempo esta característica no ha podido ser desarrollada aún a pesar de que todo el sistema de redes tenía en consideración comunicaciones a tiempo real para este tipo de partidas.

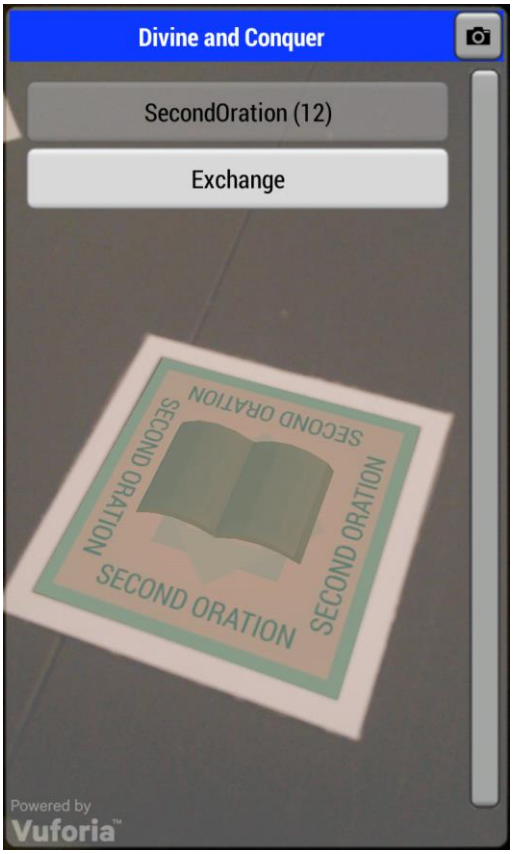
La integración de Vuforia en Unity es un *unitypackage*, que no es más que un fichero comprimido que Unity es capaz de abrir y copiar dentro del proyecto. En este fichero comprimido se incluyen las librerías nativas y las clases C# equivalentes que permiten llamar a las funcionalidades de la librería de Vuforia. Aun así, no es necesario para nada modificar el código.

Una gran baza de Unity3D es su sistema de componentes. Este permite crear componentes que se pueden incrustar a cualquier objeto que exista en una escena, y mostrar los atributos. Un componente no es más que un trozo de código enlazado con un objeto que puede modificar tal objeto y recibe eventos tales como "actualizar", el cual es ejecutado una vez por iteración del bucle del juego. El paquete de Vuforia tenía una serie de componentes que, al ser añadidos a una cámara de juego, hacían que se activase automáticamente la cámara del móvil.

La cámara de Vuforia es capaz de detectar varios markers a la vez, pero en el juego solo detectará uno. En realidad, un jugador nunca va a querer apuntar a más de una carta a la vez, explicaremos porqué más adelante. Debido a que consumía bastante batería, se ha añadido un botón en la esquina superior derecha del menú del juego que permite encender o apagar la cámara del móvil.

Cuando el modo AR está activado, el jugador puede apuntar a cualquier marker, y si es reconocido se muestra el contenido como se puede ver en las siguientes imágenes:





Ahora, en modo AR, cuando el menú está en modo juego, el juego mostrará el contenido de una carta si esta es reconocida y pertenece al jugador. En caso contrario, la carta será mostrada como

desconocida y su contenido no será revelado. Cuando se detecte una carta cambiará el menú de la siguiente manera:

- Si el jugador tiene que robar una carta y la carta no pertenece a ningún jugador, le dará la opción de robarla.
- Si es el turno del jugador y la carta detectada es una carta de su mano, mostrará al jugador dos botones. El primero permitirá usar la carta, y el segundo indicará al juego que se desea intercambiar.

Esta forma de mostrar las acciones de forma contextual hace que el juego sea más intuitivo y ágil de jugar.

Recordemos que el juego tiene 9 tipos de cartas. Cada carta del juego muestra una representación virtual diferente. La representación virtual está hecha acorde con los markers vistos en este mismo capítulo, más arriba.

En la imagen podemos ver la representación virtual de las cartas. Cada vez que se detecta una carta, el juego mira qué tipo de carta es el número detectado, y muestra la representación virtual correspondiente.

PLANIFICACIÓN Y ANÁLISIS ECONÓMICO

Para cualquier proyecto, siempre es importante hacer una valoración tanto en tiempo como en los costes asociados. Hablaremos sobre la planificación que se ha seguido durante la elaboración del proyecto y se hará una estimación del coste económico que supondría.

PLANIFICACIÓN

El proyecto, aun matriculándose en febrero, empezó sobre principios de abril. El proyecto se desarrolló en el orden en que aparecen los apartados en esta memoria. El desarrollo de Biribit, aunque se hizo a la par con la configuración de la Raspberry Pi, se alargó hasta finales de agosto. Durante esta época yo trabajaba a media jornada por lo que le dedicaba 3h de media al día al proyecto, fines de semana incluidos.

A partir de septiembre, empecé el desarrollo del juego y la integración con la realidad aumentada, llevándome a trabajar 8h de media al día. El desarrollo del proyecto finalizó a mediados de septiembre.

A continuación detallaré la metodología de desarrollo, aunque ya la haya mencionado antes brevemente. Esta está basada en métodos Agile, aunque la que he empleado en realidad es más flexible. Puesto que era un equipo de un único miembro, no había necesidad de priorizar para realizar tareas en paralelo. Las prioridades estaban basadas en qué era prerrequisito de qué. Es decir, si crear la base del cliente de Biribit requiere de incluir RakNet en el proyecto, este último era más prioritario por ser requisito.

Como hemos marcado al principio de este documento, el proyecto se compone de 4 subproyectos. Estos cuatro debían desarrollarse en el orden indicado a continuación porque los primeros eran necesarios para realizar los de abajo.

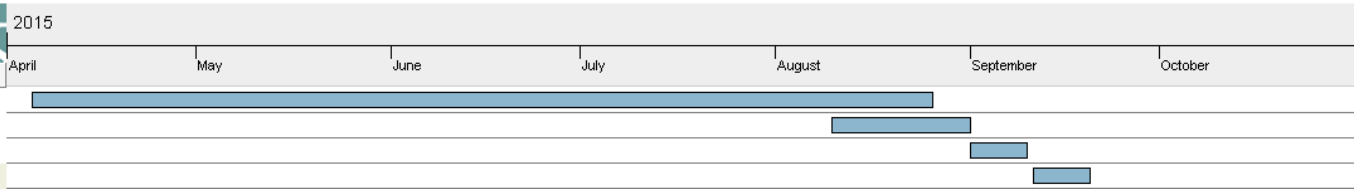
1. Biribit
2. Configurar Raspberry Pi
3. Juego Divine and Conquer
4. Integración de la Realidad Aumentada

Para cada uno de estos proyectos había un listado de funcionalidades. Realizar cada una de estas funcionalidades representaba una iteración que podían durar 2 o 3 días. El procedimiento para realizar la funcionalidad era el siguiente:

- Implementar la funcionalidad.
- Crear el juego de pruebas si es necesario, y realizar pruebas para comprobar que la funcionalidad no tiene errores.
- Hacer pruebas para ver que no se haya añadido ningún error en alguna de las funcionalidades anteriores
- Pasar a la siguiente funcionalidad.

En la imagen siguiente muestro el diagrama de Gantt de las diferentes fases del proyecto.

Name	Begin date	End date
Biribit	4/5/15	8/25/15
Configurar Raspberry	8/10/15	8/31/15
Divine And Conquer	9/1/15	9/9/15
Integrar Vuforia	9/11/15	9/19/15



ANÁLISIS ECONÓMICO

La valoración de los costes económicos se divide en dos partes: los costes de recursos humanos y los costes en material. Para calcular los costes de recursos humanos tenemos dos perfiles:

- Coste analista: 40€/h
- Coste programador: 30€/h

Se ha hecho una aproximación de horas dedicadas a cada tarea y se han asignado a su perfil respectivo.

Tarea	Perfil	Horas	Coste
Creación y diseño de la idea original del juego	Analista	10	400 €
Diseño de la arquitectura y protocolo de Biribit	Analista	50	2.000 €
Desarrollo del proyecto Biribit	Programador	400	12.000 €
Configuración de la Raspberry Pi	Programador	20	600 €
Desarrollo de la lógica del juego	Programador	25	750 €
Diseño de la experiencia de usuario del juego	Analista	5	200 €
Creación del proyecto de Unity e integración de los elementos	Programador	45	1.350 €
Integración de Realidad Aumentada	Programador	45	1.350 €
TOTAL	--	600	18.650 €

Todas las tecnologías usadas en este proyecto son o bien libres, o bien de uso gratuito, a excepción de Vuforia, que en caso de que se hiciese un uso comercial de la aplicación estaríamos obligados a hacer un pago de 499$, o lo que es lo mismo, unos 440€ al cambio.

En cuanto a los costes en material, asumiendo que cada jugador tiene su propio dispositivo móvil, los costes para tener tu juego funcional son los siguientes:

Concepto	Precio
Raspberry Pi 2	40 €
Dongle USB Wi-Fi	12 €
Power Bank 22.000 mAh	15 €
Impresión de las cartas en DinA4 en cartulina gruesa	2 €
TOTAL	69 €

Por lo tanto, asumiendo que deberíamos pagar el coste de Vuforia para hacer una explotación comercial del producto, el coste total del proyecto es de 19160€.

CONCLUSIONES Y TRABAJO FUTURO

Los objetivos principales del proyecto, como se han indicado en la introducción, eran crear un juego de cartas totalmente transportable y sin restricciones de acceso a internet. Los mini juegos, aunque fuesen prescindibles, eran parte de los objetivos iniciales y muchas decisiones de diseño en Biribit tenían en cuenta este detalle. Por lo tanto, una tarea pendiente es añadir los mini juegos.

Aun a pesar de que no se han cumplido todos los objetivos que se habían propuesto desde el principio, estoy satisfecho. Ha sido un reto para mí todo el desarrollo de Biribit, y ha sido una gran satisfacción ver cómo, una vez hechas todas las piezas, uniéndolas ha salido el juego prácticamente solo.

Otro aspecto que me gustaría explorar es llevar el juego a otras plataformas móviles, ya sean Windows Phone o iPhone. Estas plataformas fueron descartadas desde un buen principio porque requieren de hardware concreto del que no dispongo. Sin embargo, me he encontrado en varias ocasiones con las que, intentando mostrar mi juego a conocidos, no he podido hacerles disfrutar la experiencia por ser exclusivo de Android.

Por suerte, muchos de estos conocidos me han ofrecido ordenadores y dispositivos móviles de Apple para que pudiese llevar el juego a esta plataforma, por lo que no lo descarto en un futuro cercano.

Algo en lo que me gustaría profundizar es en encontrar alternativas a Raspberry Pi como punto de acceso Wi-Fi y servidor, que puedan ser más compactas y más baratas, como por ejemplo CHIP[7], un dispositivo muy pequeño con un precio de $9. Este incluye Wi-Fi, por lo que no hay necesidad de comprar un USB adicional, y además se puede comprar en una versión de $19 que incluye su propia batería. Mi deseo para este juego es que se convierta en un producto real y que sea vendido en una caja que incluya las cartas y su dispositivo de punto de acceso. Con alternativas como CHIP, esto podría ser una realidad.

También quisiera explorar la posibilidad desarrollar algún juego totalmente diferente con Biribit. En concreto, me gustaría explorar la posibilidad de poner redes Wi-Fi en lugares públicos donde la gente suela tener tiempos de espera como andenes de tren o aeropuertos, y ofrecerles la posibilidad de jugar a algún juego multijugador con las personas de su alrededor.

Y para finalizar agradezco la flexibilidad y la dedicación de mi tutor Guillem Godoy, que ha acogido desde el principio este proyecto y me ha sabido aconsejar bien para que las fechas del proyecto se cumplieran.

También quisiera agradecer a todos los amigos que me han ayudado a diseñar y mejorar las normas del juego y se han prestado muy amablemente en fases de desarrollo del juego para probarlo e ir depurando todo pequeño detalle. Gracias a ellos tengo un listado de posibles futuras normas que podrían hacer que el juego mejorase sustancialmente.

[7] http://nextthing.co/