Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO DE FIN DE CARRERA

TÍTULO DEL TFC: Implementación de una plataforma de juegos multijugador para Android

TITULACIÓN: Ingeniería Técnica de Telecomunicación, especialidad Telemática

AUTOR: Víctor Martos Egea

DIRECTOR: Sergio Machado Sánchez

FECHA: 4 de Mayo de 2011

Título: Implementación de una plataforma de juegos multijugador para Android.

Autor: Víctor Martos Egea

Director: Sergio Machado Sánchez

Fecha: 4 de Mayo de 2011

Resumen

Este documento explica el desarrollo de una plataforma para juegos multijugador destinada a ejecutarse en dispositivos con sistema operativo Android. La plataforma permitirá a móviles con sistema operativo Android poder jugar en línea entre ellos a través de wifi o utilizando la conexión de datos de los dispositivos móviles.

Se utilizará una arquitectura de red cliente/servidor, donde los clientes serán móviles Android, y tendremos dos servidores, uno destinado a la comunicación de la sala de espera y el otro a la comunicación de los juegos.

Para la sala de espera se utilizará un protocolo de comunicación de tipo publicación/suscripción desarrollado por IBM. La sala de espera permitirá a los clientes comunicarse mediante un *chat*, crear partidas y unirse a partidas ya creadas.

Se han desarrollado dos juegos diferentes, uno en tiempo real y otro por turnos. El juego en tiempo real es el Pong, y ha sido desarrollado en UDP y en TCP, para poder evaluar las diferencias entre los dos protocolos. El juego por turnos es el 3 en raya, y solamente ha sido desarrollado en TCP.

Se explicará los métodos utilizados para la sincronización de puertos en UDP, la sincronización de conexiones en TCP, el sincronismo para el juego en tiempo real, la solución para mensajes críticos en UDP, la determinación de puertos del servidor para permitir juegos simultáneos, entre otras cosas.

Title: Implementation of an online game platform for Android devices.

Author: Víctor Martos Egea

Director: Sergio Machado Sánchez

Date: May, 4th 2011

Overview

This document explains the development of a multiplayer gaming platform for devices with operative system Android. The platform will enable two or more Android devices to play online with each other via wifi or any 3G data connection system.

We have used a client/server arquitecture, where clients are Android devices, and there are two servers, one for the waiting room communication and the other for the data transport protocol of the games.

For the waiting room we have used a publish/subscribe communication protocol developed by IBM. The waiting room will allow customers to chat, create games and join to games already created.

We have developed two different games, a real-time game and a turn-based game. The real-time game is Pong, and has been developed in UDP and TCP, in order to evaluate the differences between the two protocols. The turn-based game is the 3 in a row, and only has been developed in TCP.

We will explain the methods used to synchronize UDP ports, synchronization of TCP connections, the synchronization for the game in real time, the solution for critical messages on UDP, the determination of server ports to allow simultaneous games, among other things.

ÍNDICE

INTRODUCCIÓN 1

CAPÍTULO 1. VISIÓN GENERAL 2

1.1. Arquitectura Cliente/Servidor 2
1.1.1. Esquema 2
1.1.2. Servidor *Message Broker* (MB, intermediario de mensajes) 3
1.1.3. Servidor de juego 4
1.1.4. Cliente Android 4

CAPÍTULO 2. MQTT Y SALA DE ESPERA 5

2.1. Protocolo MQTT 5
2.1.1. Formato del mensaje 6

2.2. La comunicación de la Sala de espera 10
2.2.1. Temas 11
2.2.2. Mensajes enviados 12
2.2.3. MQTT en el cliente 16
2.2.4. MQTT en el servidor de juego 17
2.2.5. Pasos de la sala de espera 18

CAPÍTULO 3. JUEGO EN TIEMPO REAL. PONG 27

3.1 Pong en UDP vs Pong en TCP 27

3.2 Protocolo de comunicación 27

3.3 Sincronismo con TimeBucket 29

3.4 Pong en UDP 30
3.4.1 Mensajes críticos 30
3.4.2 Sincronización de IP y puerto 31
3.4.3 Determinación del puerto del servidor 31
3.4.4 Estructura del código en el Cliente 32
3.4.5 Estructura del código en el Servidor 33
3.4.6 Pruebas 33

3.5 Pong en TCP 33
3.5.1 Mensaje críticos 33
3.5.2 Sincronización de conexiones 33
3.5.3 Determinación del puerto del servidor 34
3.5.4 Estructura del código en el cliente y servidor 34
3.5.5 Pruebas 35

CAPÍTULO 4. JUEGO POR TURNOS. 3 EN RAYA 36

4.1 Juegos por turnos 36

4.2 3 en raya 36

4.2.1 Protocolo de comunicación 36
4.2.2 Sincronización de la conexión 37
4.2.3 Determinar puerto servidor 37
4.2.4 Estructura del código en el cliente 37
4.2.5 Estructura del código en el servidor 39
4.2.6 Pruebas 40

CAPÍTULO 5. CONCLUSIONES Y LÍNEAS FUTURAS 41

5.1 Conclusiones 41

5.2 Ambientalización 41

5.3 Líneas futuras 42

BIBLIOGRAFIA 43

INTRODUCCIÓN

El objetivo de este documento es el diseño y la implementación de una plataforma para juegos multijugador en Android. La plataforma tiene una arquitectura de cliente/servidor. El cliente es la aplicación en Android, que se conectará a dos servidores. Primero se conectará al servidor que nos permite realizar la comunicación de la sala de espera. El segundo servidor, el servidor de juegos, se utilizará para la administración de los juegos, realizará todo lo necesario para que los clientes puedan jugar entre ellos.

El primer capítulo proporciona una visión general del proyecto desarrollado. En él se explicará detalladamente la arquitectura de red del sistema y las funciones de cada servidor.

En el segundo capitulo, se explicará el funcionamiento del protocolo de comunicación *MQ Telemetry Transport* (MQTT), desarrollado por IBM, y su implementación en la sala de espera. MQTT es un protocolo simple que permite realizar *push* a móviles utilizando un servicio.

El tercer capítulo detalla la implementación de uno de los juegos multijugador creados, el Pong. El protocolo del juego se ha desarrollado utilizando los dos protocolos estándar de la capa de transporte de TCP/IP: *User Datagram Protocol* (UDP) y *Transmission Control Protocol* (TCP). Se explicará de manera detallada el protocolo de comunicación, el sistema de sincronismo y las pruebas realizadas para el estudio de las pérdidas.

Por último, el cuarto capítulo presenta el desarrollo de otro juego multijugador, el 3 en raya. A diferencia del Pong, éste se ha desarrollado sólo sobre TCP, debido a que es un juego multijugador por turnos y no tiene sentido realizarlo en UDP. Ya que se requiere control de los mensajes para no perderlos y UDP no nos lo permite.

CAPÍTULO 1. VISIÓN GENERAL

1.1. Arquitectura Cliente/Servidor

La arquitectura de la plataforma desarrollada sigue el paradigma cliente/servidor, el más utilizado para el desarrollo de videojuegos en línea. Una estructura en P2P se podría realizar, pero al fin y al cabo se debería utilizar un servidor con IP pública que centralizaría las conexiones, sería una estructura muy parecida a la de cliente/servidor, salvo que los clientes serían los que procesarán toda la información y los cálculos en lugar del servidor. Por eso, como los móviles tienen una capacidad de procesamiento menor que la de un servidor, hemos decidido implementar la estructura de cliente/servidor.

1.1.1. Esquema

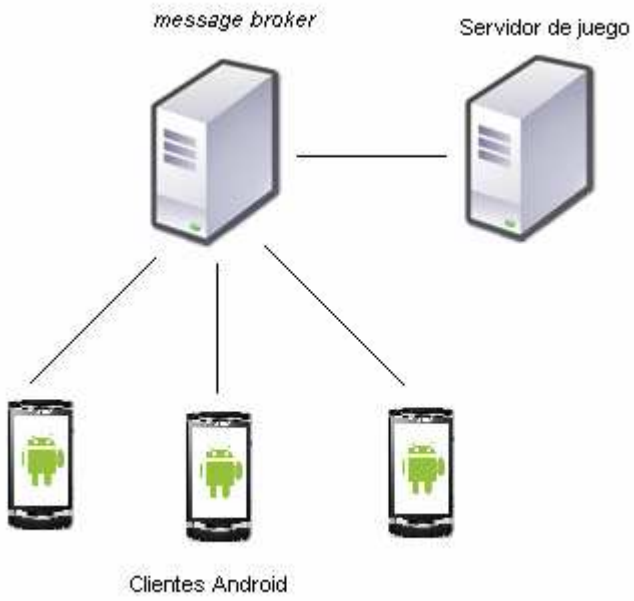

Fig. 1.1 Arquitectura de red en la Sala de Espera

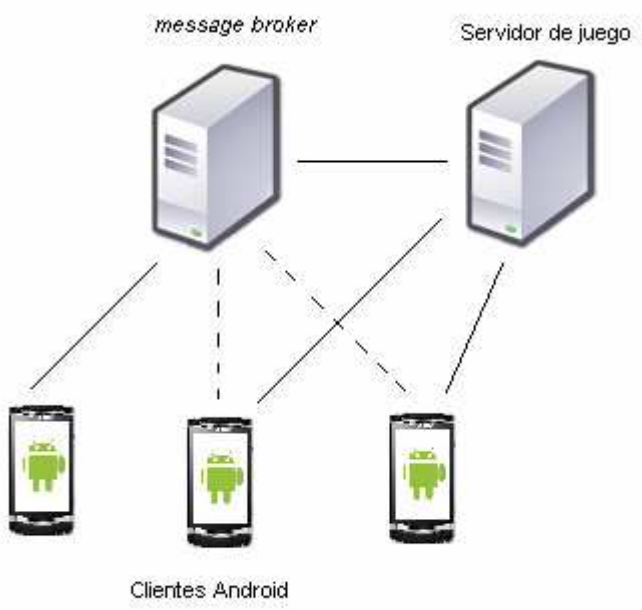

Fig. 1.2 Arquitectura de red cuando hay clientes jugando

Como se puede observar en los anteriores esquemas (Fig. 1.1 y Fig. 1.2), tenemos una arquitectura de red cliente/servidor. En concreto, la plataforma dispone de dos servidores:

- Un servidor para administrar las comunicaciones de la sala de espera
- Un servidor de juego que ha sido desarrollado para administrar la comunicación de los juegos.

Los clientes serán dispositivos móviles con sistema operativo Android. La aplicación Android está desarrollada pensando en las características del proyecto. A continuación, se explicará en detalle el funcionamiento de cada servidor y de los clientes.

1.1.2. Servidor *Message Broker* (MB, intermediario de mensajes)

Este servidor ha sido desarrollado por IBM. Tiene implementado el protocolo de comunicación MQTT que en el proyecto se utilizará para la comunicación de la sala de espera.

Concretamente el servidor es el *Really Small Message Broker* (RSMB), es un servidor de descarga gratuita que sirve para realizar proyectos de pequeñas dimensiones, ya que admite un número limitado de conexiones. Si se quiere aumentar el número de conexiones se debería contratar otro tipo de servidor de pago o realizar puentes entre varios RSMB.

Los usuarios se conectan al *message broker* y se suscriben a un cierto número de temas. Una vez suscrito a un tema, recibes los mensajes que se publicarán al tema suscrito.

1.1.3. Servidor de juego

Este servidor sirve para realizar las comunicaciones de los juegos. Hasta el momento, realiza la gestión de dos juegos diferentes: el juego del pong y el juego del 3 en raya.

El servidor también se conecta al MB para recibir los mensajes que envían los clientes Android. De esta manera se van almacenando los jugadores conectados y las partidas creadas. Cuando se conectan dos jugadores a una partida el servidor lo detecta y envía un mensaje para empezar la partida. A partir del comienzo de la partida los dos clientes Android se comunican directamente con el servidor sin intervención alguna del MB.

1.1.4. Cliente Android

Los clientes se conectan al MB y se suscriben al tema para administrar la sala de espera. Hay varios tipos de temas y mensajes para la comunicación de la sala de espera, que se explicará más adelante.

En la sala de espera los clientes utilizarán un servicio que se conectará con la aplicación de la sala de espera cada vez que reciba un mensaje.

En el juego los clientes cambiarán de actividad[1] para poder jugar y conectarse directamente con el servidor de juego; siempre sin perder la conexión con el MB.

[1] Una actividad es el interfaz de usuario que se muestra por pantalla en una aplicación. Una aplicación puede constar de varias actividades para poder interactuar con el usuario y realizar las acciones pertinentes.

CAPÍTULO 2. MQTT Y SALA DE ESPERA

2.1. Protocolo MQTT

El protocolo MQTT es un protocolo de comunicaciones usado en el paradigma publicación/suscripción. Este sistema es una manera fácil y ligera de comunicar múltiples clientes al mismo tiempo. Tiene diversos usos, al ser un protocolo ligero no es recomendable utilizarlo para situaciones donde requieran mucho tráfico de datos. Normalmente se utiliza para enviar notificaciones, monitorizar experimentos y máquinas desde casa, etc. Es un protocolo recomendable para dispositivos que no disponen de un *hardware* computacional demasiado potente.

En el protocolo MQTT todo pasa a través del MB que es el responsable de obtener los mensajes y enviarlos a los clientes, es decir, se encarga de monitorizar los mensajes. El funcionamiento es el siguiente:

1) Los clientes se conectan al MB con un nombre de usuario y se suscriben a los temas que les interesan.
2) Para enviar un mensaje el cliente lo publica, el *message broker* lo recibe y lo almacena.
3) El MB lo reenvía a los clientes correctos dependiendo del tema suscrito.
4) Si el mensaje no tiene el *flag retain* activado borra el mensaje, en caso contrario, lo sigue almacenando para futuros clientes que se suscriben al tema. Más adelante se explicará el uso del *flag retain*.

MQTT es un protocolo de tipo *push*, es decir, recibes los mensajes cuando alguien los ha publicado, sin tener que consultar al MB. La tecnología *push* es muy útil para móviles, ya que no requiere de un sistema para consultar si hay o no mensajes. Simplemente cuando un usuario se conecta al MB y se suscribe a un tema y el MB tiene un mensaje del tema suscrito, éste se lo envía directamente al móvil del usuario haciendo *push*.

El nombre de usuario que se conecta al MB debe ser único, ya que si se conecta un usuario con un nombre que ya esta siendo utilizado por otro usuario, el MB desconectará automáticamente a éste último, es decir, al jugador que llevaba más tiempo conectado. Este problema se evita consultando primero al servidor de juego si el usuario ya está conectado al MB o no.

A continuación se explicará la estructura del protocolo MQTT para que tengáis una breve idea de cómo funciona, no vamos a entrar en detalle, ya que se va a explicar de manera general el protocolo.

2.1.1. Formato del mensaje

Todos los mensajes tienen la misma cabecera (*Fixed Header*) dicha cabecera consta de los siguientes campos:

Tabla 2.1. Cabecera fija de los mensajes MQTT.

bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	Remaining Length							

A continuación, en los siguientes apartados se detallará el significado de cada campo y para qué se utiliza.

2.1.1.1. Tipo de mensaje (Message Type)

La siguiente tabla relaciona los tipos de mensajes con su numeración y su descripción:

Tabla 2.2. Tipos de mensajes enviados en MQTT

Mnemonic	Enumeration	Description
Reserved	0	Reserved
CONNECT	1	Client request to connect to Server
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received (assured delivery part 1)
PUBREL	6	Publish Release (assured delivery part 2)
PUBCOMP	7	Publish Complete (assured delivery part 3)
SUBSCRIBE	8	Client Subscribe request
SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	PING Request
PINGRESP	13	PING Response
DISCONNECT	14	Client is Disconnecting
Reserved	15	Reserved

La tabla muestra todos los mensajes que componen el protocolo MQTT. Los más importantes son CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE y

DISCONNECT. Los demás son mensajes de reconocimiento (CONNACK, PUBACK, SUBACK, UNSUBACK) o mensajes relacionados con la calidad de servicio de los mensajes PUBLISH (PUBACK, PUBREL, PUBREL y PUBCOMP).

2.1.1.2. DUP flag

DUP viene de duplicado, este *flag* se utiliza cuando el cliente o el servidor intentan reenviar un mensaje PUBLISH, PUBREL, SUBSCRIBE o UNSUBSCRIBE. Se aplica a mensajes cuando la calidad de servicio (*QoS*) es mayor que 0, o se requiere un reconocimiento del mensaje.

2.1.1.3. Calidad de servicio (QoS)

El servicio MQTT entrega los mensajes de acuerdo a los niveles definidos de la calidad del servicio en el mensaje de tipo PUBLISH. Indica si requiere o no recibir reconocimiento de los mensajes que se han enviado. Los niveles se explican a continuación:

Tabla 2.3. Tabla resumen de los diferentes niveles de QoS

QoS value	bit 2	bit 1	Description		
0	0	0	At most once	Fire and Forget	<=1
1	0	1	At least once	Acknowledged delivery	>=1
2	1	0	Exactly once	Assured delivery	=1
3	1	1	Reserved		

QoS* level 0: *At most once delivery

Con este nivel no se hará reconocimiento del mensaje a nivel del protocolo MQTT. Se enviará el mensaje y el cliente se olvidará de él. Un nivel no óptimo cuando se requiera de un sistema sin pérdidas.

Tabla 2.4. Mensaje PUBLISH con *QoS* = 0

Client	Message and direction	Server
QoS = 0	PUBLISH ---------->	**Action:** Publish message to subscribers

QoS* level 1: *At least once delivery

Este nivel de calidad de servicio permite un reconocimiento del mensaje. El servidor enviará al cliente una confirmación de mensaje recibido. Si el cliente no recibe el PUBACK, enviará otra vez el mensaje hasta recibirlo. Esto permite saber que el servidor ha recibido correctamente el mensaje. Si se pierde el

mensaje PUBACK (reconocimiento del mensaje PUBLISH) el cliente volverá a enviar el mensaje PUBLISH. Por lo tanto, el servidor recibirá el mensaje PUBLISH dos veces.

Tabla 2.5. Mensaje PUBLISH con QoS = 1

Client	Message and direction	Server
QoS = 1 DUP = 0 Message ID = x **Action:** Store message	PUBLISH ---------->	**Actions:** • Store message • Publish message to subscribers • Delete message
Action: Discard message	PUBACK <----------	

QoS level 2: *Exactly once delivery*

Esta calidad de servicio consiste en un doble reconocimiento. El cliente publica un mensaje PUBLISH, el servidor se lo guarda y envía un PUBREC diciendo que ha recibido el mensaje. Acto seguido, el cliente enviará un PUBREL pidiendo que se publique el mensaje. El servidor una vez publicado, enviará un mensaje PUBCOMP al cliente diciéndole que ya se ha enviado. En este nivel, a diferencia del nivel 1, sólo se enviará una vez el mensaje, ya que el *Message ID* será el mismo en todos los intentos. Por lo tanto, si ha enviado el PUBLISH con un *Message ID* "x", y el PUBREC no lo recibe, volverá a enviar el PUBLISH con el mismo M*essage ID*. El servidor, al ver que es el mismo *Message ID*, sabe que el cliente no ha recibido el PUBREC anterior y vuelve a enviar otro.

Tabla 2.6. Mensaje PUBLISH con QoS = 2

Client	Message and direction	Server
QoS = 2 DUP = 0 Message ID = x **Action:** Store message	PUBLISH ---------->	**Action:** Store message *or* **Actions:** • Store message ID • Publish message to subscribers
	PUBREC <----------	Message ID = x
Message ID = x	PUBREL ---------->	**Actions:** • Publish message to subscribers • Delete message *or* **Action:** Delete message ID
Action: Discard message	PUBCOMP <----------	Message ID = x

2.1.1.4. RETAIN

El *flag* sólo se utiliza para los mensajes de tipo PUBLISH. Si se envía un mensaje PUBLISH a un tema con el *flag retain* activado, el MB se guardará el mensaje para poder enviarlo a los usuarios que se suscriban más tarde al tema. Por lo tanto, da un grado más alto de importancia al mensaje, ya que se lo enviará a cada cliente que se vaya suscribiendo a ese tema. Por eso, el *flag retain* es muy útil para la sala de espera que hemos desarrollado, ya que nos servirá para que los usuarios nuevos que se vayan conectando reciban todos los mensajes cuando se conecten, y por lo tanto actualizarán la sala de espera.

Se pueden tener tantos mensajes *retain* como temas diferentes. No se pueden tener dos mensajes *retain* sobre un mismo tema. Por lo tanto, si se envía un mensaje con el *flag retain* a un tema que ya tenía un mensaje guardado, el mensaje será sustituido por éste último. Los usuarios que se suscriban al tema recibirán sólo el mensaje más nuevo.

Para hacer que el servidor elimine un mensaje *retain* de un tema, se deberá enviar un mensaje *retain* al mismo tema con un *payload length* (tamaño de datos) de 0. Ésta opción la usaremos en un par de ocasiones en la sala de espera.

2.1.1.5. Remaining length

Representa el número de bytes que restan del mensaje, incluyendo los datos de la cabecera variable y el *payload*.

2.1.1.6. Temas

Para que un mensaje se pueda enviar necesita un tema, y para que un usuario pueda recibir mensajes necesita estar suscrito a un tema. Así de sencillo es el sistema de funcionamiento por temas de MQTT.

Las suscripciones pueden contener caracteres especiales, que te permiten suscribirte a múltiples temas a la vez y se pueden estructurar de manera jerárquica:

- **Topic level separator**

  La barra diagonal (/) se utiliza para separar cada nivel dentro de un árbol de temas y ofrecer una estructura jerárquica. Veámoslo con un ejemplo de los temas utilizados en la sala de espera:

  3 en raya/usuarios

  3 en raya/partidas

  3 en raya/Chat

En este ejemplo podemos ver la estructura que tendrá la sala de espera, usuarios, partidas y chat pertenecen al tema 3 en raya, pero son temas independientes.

- ***Multi-level wildcard***

  El signo almohadilla (#) es un carácter comodín que coincide con cualquier número de niveles dentro de un tema. Por ejemplo, si se suscribe a 3 en raya/# recibe mensajes en los siguientes temas:

  3 en raya/usuarios/nombre_usuario

  3 en raya/usuarios

  3 en raya/partidas/nombre_partida/nombre_usuario

  3 en raya/chat

  3 en raya

- ***Single-level wildcard***

  El signo más (+) es un carácter comodín que coincide con un solo nivel de la jerarquía de temas. Por ejemplo, si se suscribe a 3 en raya/+ recibe mensajes de los siguientes temas:

  3 en raya/usuarios

  3 en raya/partidas

  3 en raya/chat

  Pero no recibes mensajes de:

  3 en raya

  3 en raya/partidas/nombre_partida

  3 en raya/usuarios/nombre_usuario

2.2. La comunicación de la Sala de espera

Se ha pensado que nuestra sala de espera sea desarrollada en MQTT, porque dicho protocolo permite estar todo el rato conectado y recibir los mensajes de la conexión de los usuarios, de la creación de partidas y de los mensajes de chat.

Para que un juego multijugador tenga éxito se debe tener una sala de espera donde se podrán ver los usuarios que hay conectados, se podrán crear partidas y también podrán comunicarse mediante *chat*. Esto último es muy importante, ya que al permitir la comunicación entre usuarios también permites que conozcan a gente y jueguen entre ellos. Una se ha sabido para que se utilizará

la sala de espera, se ha procedido al diseño de los temas, que se pueden ver a continuación.

2.2.1. Temas

Para la comunicación de la sala de espera se utilizará el siguiente esquema de temas, habrá tres raíces:

1. Juego/usuarios/#
2. Juego/partidas/#
3. Juego/chat/#

Como se puede observar no hay tres temas diferentes, ya que como se ha explicado antes la '#' indica que el tema sigue. En este proyecto se ha pensado en tener tantos temas como usuarios hayan conectados. Eso es debido a que donde se encuentra el comodín '#' se pondrá el nombre de usuario conectado o el nombre de la partida.

Para la gestión de los usuarios se utilizará el tema 1) Juego/usuarios/#, donde # será el nombre de usuario del jugador.

Para la gestión de las partidas, se utilizará el tema 2) Juego/partidas/#, donde # será el equivalente al nombre de la partida/usuario que realiza la acción de crear o unirse a la partida.

Para el chat de la sala de espera se publicarán los mensajes bajo el tema 3) Juego/chat/#, donde # será el nombre del usuario que envía el mensaje de chat.

Para que se entienda mejor vamos a ver varios ejemplos:

Ej. 1) El usuario Victor se conecta a la sala de espera del juego Pong, por lo tanto enviará un mensaje al tema: "Pong/usuarios/Victor" informando que se ha conectado.

Ej. 2) El usuario Victor crea una partida llamada Partida1 en el juego Pong. La aplicación publicará un mensaje con el tema: "Pong/partidas/Partida1/Victor" informando que se ha creado una partida.

Ej. 3) El jugador Victor saluda a los demás usuarios (en el juego del Pong) enviando un mensaje de chat. Para ello, la aplicación enviará el mensaje de chat bajo el tema "Pong/chat/Victor".

Para que los usuarios reciban los mensajes, se suscribirán al siguiente tema: "Juego/#". No hace falta suscribirse a los tres temas raíces gracias al comodín "#".

2.2.2. Mensajes enviados

Al principio se diseñó la sala de espera en TCP, por lo que cada mensaje debía contener el nombre de usuario, el nombre de la partida a crear, etc. Con el sistema de temas, los mensajes se han hecho mucho más sencillos y cortos. Ahora aprovechamos el tema para obtener información, por ese motivo los mensajes únicamente tendrán como valor un entero, que será el tipo de mensaje. Excepto en contadas ocasiones donde se necesite más información.

Los mensajes que se enviarán son los siguientes:

- Mensaje del cliente para comprobar el nombre de usuario (C_NICK):

  Este mensaje se envía en TCP, ya que se tiene que hacer una comprobación antes de conectarte al MB para saber si el usuario con el que te quieres conectar está ya conectado. Por lo tanto, el mensaje ha de ser de un protocolo distinto al MQTT.

  Los datos del mensaje enviado:

  - int correspondiente a C_NICK: 19
  - String con el nombre del usuario a comprobar

- Mensaje de respuesta a la comprobación del usuario (C_NICK_R):

  Este mensaje lo envía el servidor al cliente como respuesta al mensaje C_NICK. El servidor una vez realizada la comprobación, responde con el siguiente mensaje:

  - int correspondiente a C_NICK_R: 22
  - Boolean: true → jugador existe
    false → jugador no existe

- Mensaje para la conexión del usuario (LOGIN):

Tabla 2.7. Estructura del mensaje LOGIN

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_usuario"	3 (int)	2	true

Este mensaje se envía cuando se conecta al MB, con este mensaje se está indicando a los demás usuarios que se ha conectado, para que puedan actualizar su sala de espera. Tiene el *retain* activado, por lo tanto, cada vez que un nuevo usuario se conecte al MB y se suscriba al tema de la sala de espera, recibirá el mensaje LOGIN de todos los usuarios que hay conectados.

- Mensaje de desconexión del usuario (LOGOUT):

Tabla 2.8. Estructura del mensaje LOGOUT

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_usuario"	15 (int)	1	false

Se envía cuando un usuario se desconecta de la sala de espera y del MB. Éste será el único mensaje (junto con el 0 *byte length* de desconexión) que se enviará con QoS 1. Esto es debido, a que justo después de enviar el mensaje se desconecta del MB. Por lo tanto, si fuera un mensaje QoS 2 no podría realizar el doble reconocimiento y el mensaje no se publicaría y los usuarios no recibirían el mensaje de desconexión y se pensarían que aún esta conectado.

- Mensaje publicado cuando un usuario crea una partida (CREATE):

Tabla 2.9. Estructura del mensaje CREATE

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_partida/nombre_usuario"	13 (int)	2	true

Al igual que en el mensaje LOGIN, el *retain* está activado, ya que nos interesa que cuando se conecte un usuario nuevo reciba el mensaje de la creación de partida para que actualice su sala de espera y se sincronice con todos los usuarios.

- Mensaje para cancelar la partida (CANCEL_GAME):

Tabla 2.10. Estructura del mensaje CANCEL_GAME

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_partida/nombre_usuario"	17 (int)	2	false

Se envía cuando un usuario quiere cancelar su partida creada.

- Mensaje para unirse a una partida seleccionada (JOIN):

Tabla 2.11. Estructura del mensaje JOIN

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_partida/nombre_usuario"	14 (int)	2	true

Se envía cuando un usuario quiere unirse a una partida. Tiene el *retain* activado por el mismo motivo que en los casos anteriores.

- Mensaje que se publica cuando se acaba una partida (GAME_FINISHED):

Tabla 2.12. Estructura del mensaje GAME_FINISHED

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_partida "	20 (int)	2	false

El mensaje lo publica el creador de la partida cuando ésta ha terminado.

- Mensaje que se publica para que empiece una partida (START):

Tabla 2.13. Estructura del mensaje START

TEMA	DATOS			FLAGS	
				QoS	Retain
"Juego/usuarios/nombre_partida "	2 (int)	Jugador (int)	Numero partida (int)	2	false

El mensaje lo publica el servidor de juego. Los dos jugadores (el creador y el que se ha unido) lo reciben y ejecutan otra actividad para poder jugar.

Los datos del mensaje están compuestos por tres números enteros. El primer entero es el tipo de mensaje, que se utiliza para diferenciar todos los mensajes. El segundo entero es el número de jugador en el juego. Normalmente el 0 será el creador y el 1 el que se une a la partida. Y finalmente el número de partida servirá para determinar el puerto que se utilizará para esa partida en concreto.

- Mensaje para el *chat*:

Tabla 2.14. Estructura del mensaje del chat

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/chat/nombre_usuario"	String *chat*	2	false

Este mensaje se envía cada vez que un usuario se comunica mediante el sistema de *chat*. A diferencia de todos los mensajes anteriores, únicamente se enviará un *String* que contiene el mensaje que el usuario ha escrito antes de darle a enviar.

- Mensaje “0 *payload length*”:

Este mensaje se enviará cada vez que se quiera eliminar un mensaje *retain* en un tema específico. En total se enviará en tres ocasiones:

1. Cuando se desconecta un usuario.
2. Cuando se cancela una partida.
3. Cuando se elimina una partida acabada.

En el primer caso, servirá para eliminar el mensaje LOGIN, que era *retain*, del usuario que se desconecta. De esta manera, si un usuario nuevo se conecta, no recibirá la publicación del mensaje LOGIN del usuario desconectado. La estructura del mensaje es la siguiente:

Tabla 2.15. Estructura del mensaje “0 *payload length*” para la desconexión de usuario

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_usuario"	-	1	true

Tanto el segundo como el tercer caso servirán para eliminar el mensaje *retain* de creación de la partida que había guardado el MB. Por lo tanto, si se conecta un usuario nuevo no recibirá el mensaje CREATE. El mensaje será el mismo en ambos casos, y lo enviará el creador de la partida cuando éste la cancela o acaba de jugar la partida:

Tabla 2.15. Estructura del mensaje “0 *payload length*” para la eliminación de una partida

TEMA	DATOS	FLAGS	
		QoS	Retain
"Juego/usuarios/nombre_partida/nombre_usuario "	-	2	true

2.2.3. MQTT en el cliente

Para la administración de la sala de espera en Android se dispone de una actividad y de un servicio. Que están relacionados entre sí para poder comunicarse y realizar las acciones pertinentes.

2.2.3.1. Servicio MQTTService

Un servicio es una aplicación que corre en *background*. No es visible por parte del usuario. Lo que hace este servicio es conectarse al MB y suscribirse al tema del juego.

Cada vez que el servicio recibe un mensaje del MB lo reenvía a la actividad con la función: *broadcastReceivedMessage (topic, payloadbytes);* y la actividad lo recibe con el *messageReciever*.

El servicio también puede avisar al usuario (si éste no tiene la actividad de la sala de espera activa) cuando llega un mensaje mediante notificaciones. Pero dicha opción no se ha implementado, ya que no lo hemos visto necesario.

2.2.3.2. Actividad SalaEspera

La actividad de la sala de espera es la interfaz de usuario que gestiona la conexión de los usuarios y la creación de partidas. La actividad recibe los mensajes que le reenvía el servicio y los guarda en un *array* de usuarios y en un *array* de partidas. De esta manera, se muestra por pantalla los jugadores conectados y las partidas creadas o en juego.

En la sala de espera hay un botón para crear una partida. Cuando creas una partida se te abre otra actividad para poner el nombre de la partida. Una vez puesto el nombre, el cliente publica un mensaje para que los demás usuarios actualicen la sala de espera.

También se dispone de un sistema de *chat*, para ello se tiene un campo de texto y un botón para enviar a los demás usuarios el texto escrito.

2.2.4. MQTT en el servidor de juego

El servidor de juego también tiene que estar escuchando constantemente los mensajes que envían los usuarios, ya que cuando se conectan dos usuarios a la misma partida el servidor lo detecta y publica en el tema “Juego/partidas/nombre partida” un mensaje START. De esta manera, los jugadores reciben el mensaje START y empieza la partida.

Además, el servidor de juego al estar escuchando todos los mensajes va guardando los usuarios que están conectados y las partidas en variables. De esta manera podemos realizar la comprobación de si un usuario ya está conectado con el mismo nombre, para así evitar que el MB desconecte al jugador que ya estaba conectado como hemos explicado anteriormente. Para ello, se quedará a la escucha de conexiones TCP en el puerto 20000. Una vez recibido el tipo de mensaje C_NICK, mencionado anteriormente, comprobará si el usuario está ya conectado y le contestará con el tipo de mensaje C_NICK_R

para indicarle si el usuario está ya en uso o no. La estructura exacta de los mensajes C_NICK y C_NICK_R la podéis ver en el apartado 2.2.2.

El servidor de juego cuando empieza la partida crea un *thread* nuevo para realizar la comunicación con los jugadores y seguir escuchando los mensajes de la sala de espera. De esta manera, el servidor puede gestionar varias partidas simultáneas.

2.2.5. Pasos de la sala de espera

A continuación se explicará de manera detallada los mensajes que publica y las variables que actualiza la sala de espera cuando los usuarios se conectan, crean partidas, se unen, etc.

Para ello vamos a utilizar tres usuarios:

- Usuario 1, con *nickname* Victor, será el creador de la partida.
- Usuario 2, con *nickname* Pepe, será el que se unirá a la partida.
- Usuario3, con *nickname* jugador, para comprobar que si se está jugando una partida no se puede unir.

Cada usuario deberá abrir la aplicación instalada, que se llama *Android Games*. Una vez abierto deberán seleccionar Mutlijugador para poder jugar en línea con otros jugadores.

Fig 2.1. Pantalla principal de nuestra aplicación

Este proyecto esta pensado para juegos multijugador en línea, aunque se puede observar en la figura 2.1 que hay una opción para poder jugar individual. En este modo de juego, por ahora, solo se podrá jugar al Pong.

Dentro de multijugador se abrirá otra actividad para seleccionar a qué juego quieres jugar. En este ejemplo seleccionaremos PONG UDP.

Fig. 2.2. Pantalla para la selección del juego

Una vez seleccionado el juego nos pide la IP del servidor. Dicha IP se deberá escribir, ya que de momento está en periodo de pruebas. Una vez se publique la aplicación al *android market* se debería obtener una IP pública y un servidor para colgar allí el MB y el servidor de juego. Ésta actividad como se puede observar en la siguiente figura 2.3 también pide el *nickname*.

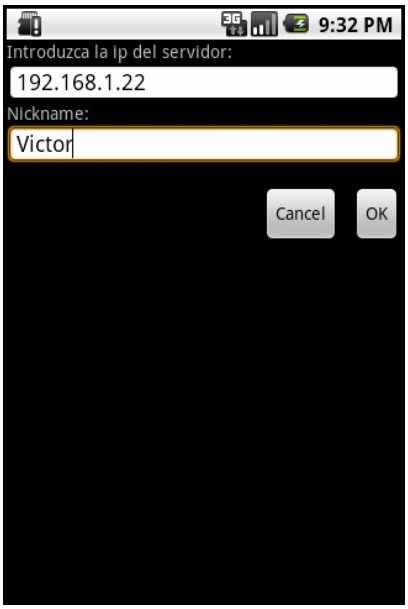

Fig. 2.3. Pantalla para determinar la IP y el nombre de usuario

Una vez introducido el *nickname* y la IP, la actividad comprueba primero si ese *nickname* ya está siendo utilizado. En concreto, se envía un mensaje de tipo C_NICK que tendrá los siguientes datos:

- 19 (int)
- Victor (String)

El servidor responderá con un mensaje C_NICK_R, con un *boolean* indicando *false* si el usuario no existe, por lo tanto, te puedes conectar. Se indicará *true* si el usuario existe y por lo tanto se mostrará por pantalla el siguiente mensaje:

Fig. 2.4. Error al conectarte con un usuario que ya está conectado

Si se conectan con éxito pasaremos a la sala de espera y se enviarán los respectivos mensajes LOGIN de cada usuario al tema correspondiente y con el *flag retain* activado.

Fig 2.5. Sala de espera con el usuario Victor conectado

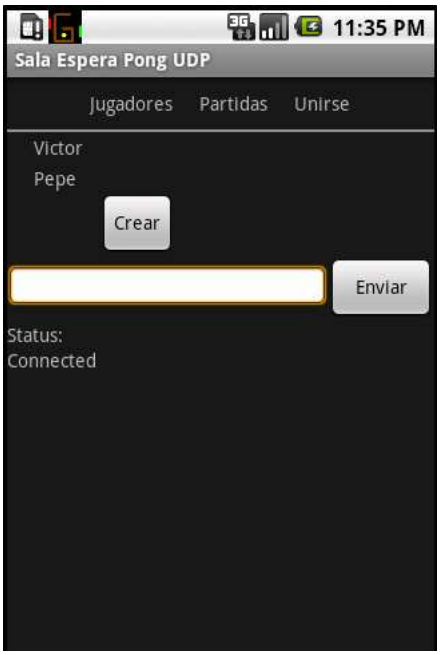

Fig. 2.6. Sala de espera con los usuarios Victor y Pepe conectados.

Cuando se conecta el primer usuario verá su nombre en la columna de usuarios. Cuando el siguiente usuario se conecta recibe la publicación LOGIN de dicho usuario y automáticamente se actualiza la columna de usuarios, como se ha podido observar en las anteriores figuras.

Cada usuario podrá publicar mensajes de *chat* para comunicarse con los usuarios que hay conectados. Lo podemos ver a continuación:

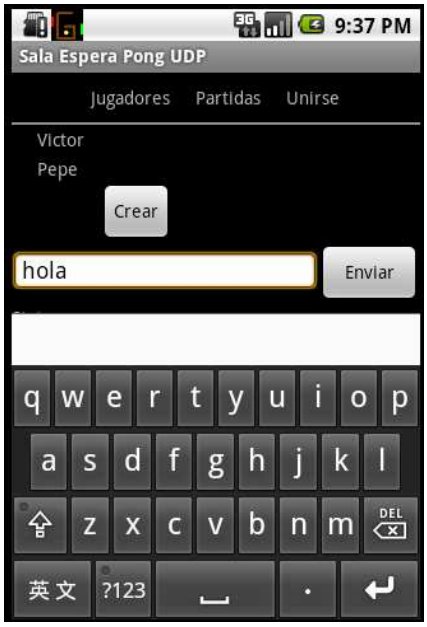

Fig. 2.7. El usuario Victor está escribiendo un mensaje de chat

Fig. 2.8. El usuario Pepe responde al chat enviado por Victor

Los usuarios en cualquier momento pueden crear una partida. Para crearla se deberá indicar el nombre de la partida en otra Activity. Una vez puesto el nombre el usuario publica un mensaje CREATE.

Cuando los demás usuarios reciben la publicación del mensaje actualizan su sala de espera del mismo modo que cuando se conectan los usuarios. En este ejemplo que estamos detallando, el usuario Pepe crea la partida y envía un mensaje CREATE al tema “Pong UDP/partidas/Partida1/Pepe” con el *flag retain* activado y los dos usuarios ven exactamente la figura que hay a continuación:

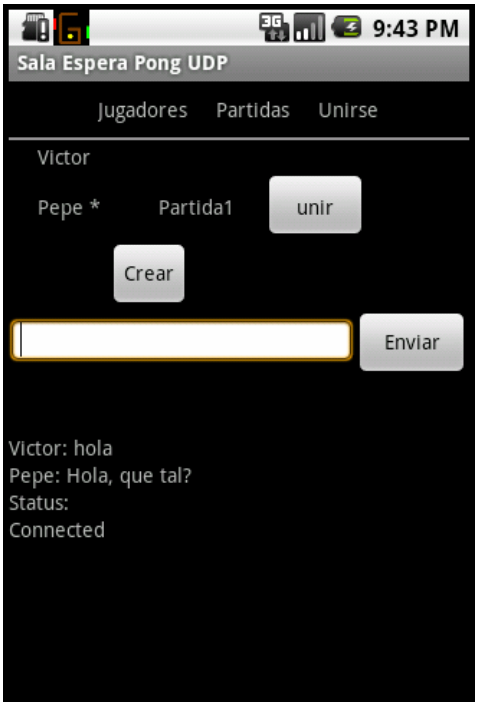

Fig. 2.9. El usuario Pepe ha creado una partida

Nótese que el nombre de la partida se muestra en la columna de partidas y se muestra al lado del usuario que ha creado la partida. También se puede observar que se crea un botón de unir para que los usuarios puedan unirse a dicha partida.

También está la opción de cancelar la partida, se puede cancelar apretando al botón de cancelar o bien apretando la tecla de retroceso. En este caso se publicará un mensaje CANCEL_GAME al siguiente tema “Pong UDP/partidas/Partida1/Pepe”. Seguidamente también se deberá enviar el mensaje de 0 payload length al siguiente tema “Pong UDP/partidas/Partida1/Pepe” con el *flag retain* activado.

Los usuarios cuando reciben el CANCEL_GAME lo procesan y eliminan la partida de la sala de espera.

Ahora el jugador Victor se dispone a unirse a la partida para poder jugar, por lo tanto, deberá apretar el botón unir. El jugador Victor publicará un mensaje JOIN al tema “Pong UDP/partidas/Partida1/Victor” con el *flag retain* activado:

El mensaje será escuchado tanto por los usuarios como por el servidor de juego. Los usuarios verán los asteriscos que indican que la partida está en juego y además se les prohibirá unirse tal y como muestra la siguiente figura (Fig. 2.10.). Para realizar esto se ha conectado un tercer usuario, con nombre jugador, mientras Victor y Pepe están jugando:

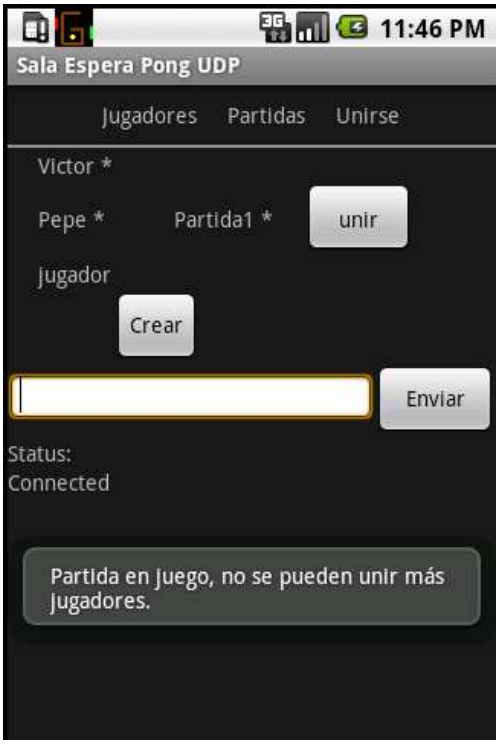

Fig. 2.10. El usuario jugador intenta conectarse a una partida en juego

Por otro lado, el servidor cuando recibe el mensaje de unir significa que ya hay dos jugadores en dicha partida, por lo que publicará un mensaje START al tema “Pong UDP/partidas/Partida1”:

El servidor iniciará un *thread* nuevo para gestionar las comunicaciones del juego (que veremos en el siguiente capítulo).

Todos jugadores recibirán el mensaje pero sólo los que se han unido lo procesarán e iniciarán la actividad del juego del pong en UDP. Como se puede observar en las siguientes figuras (Fig. 2.11. Y Fig. 2.12.):

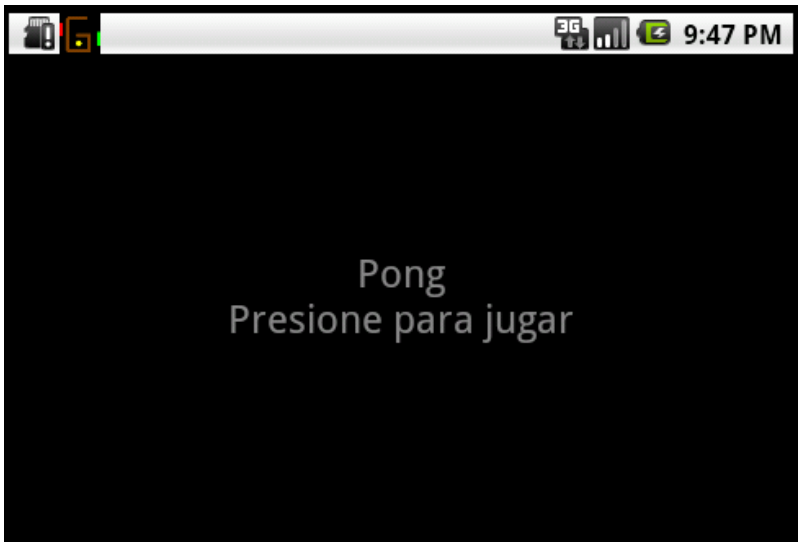

Fig. 2.11. Inicio del juego del Pong

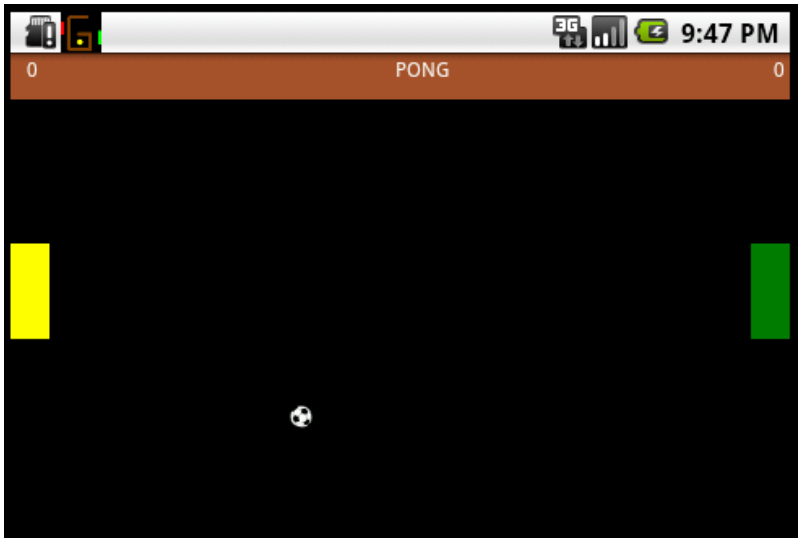

Fig 2.12. Pantalla de los usuarios jugando al juego del Pong

Cuando el juego acaba se abre otra vez la aplicación de la sala de espera, el usuario que ha creado la partida envía un mensaje GAME_FINISHED al tema “Pong UDP/partidas/Partida1” y un mensaje 0 byte length al tema “Pong UDP/partidas/Partida1/Pepe” con el *flag retain* activado para eliminar el anterior mensaje *retain*. El usuario Victor que fue el que se unió a la partida, sólo enviará un mensaje 0 byte length al tema “Pong UDP/partidas/Partida1/Victor” con el *flag retain* activo para eliminar el mensaje JOIN, que también era *retain*.

Una vez acabada la partida, se puede observar que después de enviar el mensaje GAME_FINISHED se elimina la partida de la sala de espera:

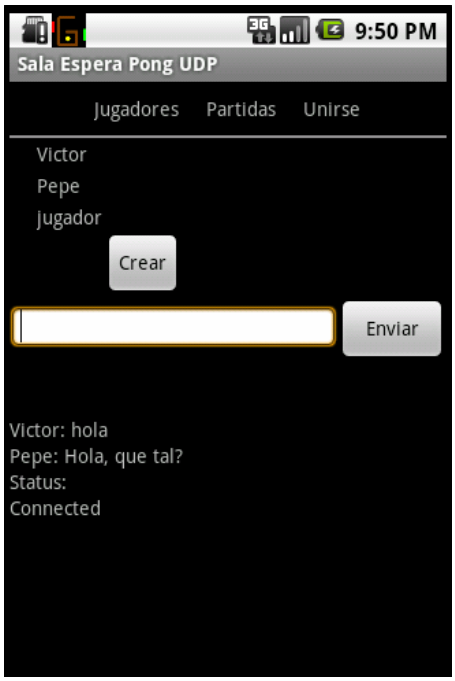

Fig. 2.13. Al terminar la partida los usuarios vuelven a la sala de espera y se elimina la partida de la lista

En la sala de espera los jugadores pueden desconectarse siempre que quieran, para ello solo se tiene que apretar la tecla de retroceso. Se volverán a enviar dos mensajes:

1) El mensaje de LOGOUT al tema del usuario correspondiente.
2) El mensaje 0 byte length con el *flag retain* activado, para eliminar el mensaje LOGIN del MB, ya que era *retain*.

CAPÍTULO 3. JUEGO EN TIEMPO REAL. PONG

3.1 Pong en UDP vs Pong en TCP

El juego del pong es el juego electrónico más antiguo que existe, consiste en evitar que la bola sobrepase límite de tu campo, para ello se dispone de una pala para hacer que la bola rebote y enviársela al adversario.

Se ha diseñado el juego del pong en dos protocolos de transporte distintos, en UDP y en TCP. La diferencia principal de ambos protocolos es que en UDP no hay un reconocimiento de los mensajes, por lo que es si la red falla no se bloqueará la conexión con reconocimientos. Eso hace que para juegos en tiempo real sea más apropiado utilizar UDP en lugar de TCP. Pero visto que el juego del pong no requiere de muchas transmisiones de datos, se ha pensado que en TCP se podría obtener una jugabilidad óptima.

3.2 Protocolo de comunicación

Los mensajes enviados en UDP y TCP serán los mismos, ya que obviamente es el mismo juego. Únicamente, cambiará la sincronización inicial, ya que TCP es un protocolo orientado a conexión, y no necesitará de mensajes iniciales para determinar la IP y puerto de los jugadores. En cambio, en UDP se deberán enviar algunos mensajes previos al inicio del juego, ya que el servidor necesita saber las IP's y puertos que utilizarán los jugadores antes de empezar a enviar los datos de la bola, puntuaciones, etc.

El servidor será el encargado de realizar todas las comprobaciones del juego, es decir, se encargará del movimiento de la bola, de las colisiones con las palas de los jugadores y de realizar los *pauses* y *unpauses* enviados por el cliente. Por lo tanto, el servidor será el encargado de enviar la posición de la pelota, la puntuación y posición de los jugadores y de reenviar el *pause* o *unpause*.

El cliente, en cambio, sólo se encargará de mover la pala que le corresponda y de pausar la partida en caso de necesitarlo. Por lo tanto, los mensajes que se enviarán serán el de la posición de la pala y el mensaje de *pause* o *unpause*.

Los mensajes que se envían están compuestos de los datos necesarios para poder actualizar el juego. No todos los mensajes tienen los mismos campos, ya que para hacer un *pause* del juego, no se necesitará enviar la posición de la pelota. Para poder diferenciar el mensaje y poder recibirlo correctamente, el primer dato del mensaje siempre será un número entero indicando qué tipo de mensaje es, es decir, si se trata de un mensaje de la posición de la bola, un mensaje de posición de pala, etc.

Se ha implementado un protocolo de sincronismo de tipo *timebucket*. Tanto el servidor como el cliente se conectan a un servidor *Network Time Protocol* (NTP) para calcular el *offset* de su reloj interno con el reloj NTP del servidor. De

esta manera todos tienen el mismo reloj y se puede hacer una sincronización óptima. Por lo tanto, el segundo campo de cada mensaje será el *timestamp*, necesario para saber exactamente a qué instante de tiempo se tendrá que ejecutar el mensaje enviado. Más adelante se explicará en detalle el funcionamiento del *timebucket*.

Otro campo necesario en este juego y en la mayoría de juegos en línea, es el campo jugador. Esto es debido a la necesidad de diferenciar el jugador que ha realizado la acción. Como el juego del pong es de dos personas, el campo de jugador únicamente podrá ser 0 o 1.

A continuación se detalla todos los mensajes que se enviarán con la información que contienen:

Mensajes enviados por el servidor

- Mensaje BALL:

  - Tipo: 1 (int)
  - *Timestamp* (long)
  - Posición de la bola x (int)
  - Posición de lo bola y (int)

- Mensaje MOVE:

  - Tipo: 0 (int)
  - *Timestamp* (long)
  - Jugador (int)
  - Posición y del jugador (int)

- Mensaje SCORE:

  - Tipo: 5 (int)
  - *Timestamp* (long)
  - Jugador (int)
  - Puntuación (int)

- Mensaje WIN:

  - Tipo: 11 (int)
  - *Timestamp* (long)
  - Jugador (int)

- Mensaje PAUSE:

  - Tipo: 8 (int)
  - *Timestamp* (long)
  - Jugador (int)

- Mensaje UNPAUSE:

  - Tipo: 10 (int)
  - *Timestamp* (long)
  - Jugador (int)

- Mensaje EXIT_GAME:

  - Tipo: 18 (int)
  - *Timestamp* (long)
  - Jugador (int)

Mensajes enviados por el cliente:

- Mensaje MOVE
- Mensaje PAUSE
- Mensaje UNPAUSE
- Mensaje EXIT_GAME

El cliente únicamente envía estos cuatro mensajes. Los campos son idénticos a los mensajes del mismo tipo que envía el servidor, por eso no los hemos indicado.

Los mensajes MOVE, PAUSE, UNPAUSE y EXIT_GAME los genera el cliente y el servidor se encarga de procesarlos y reenviarlos al otro cliente para que pueda refrescar correctamente la pantalla.

Como se puede observar, el servidor envía muchos más mensajes que el cliente, esto es debido a que todos los cálculos y procesos los ejecuta el servidor y también reenvía los mensajes que envía un cliente a otro.

3.3 Sincronismo con *TimeBucket*

El sincronismo es muy importante en un juego en tiempo real, se necesita una manera para que todos los jugadores vean lo mismo en el mismo instante de tiempo. De esta manera, se evita que un jugador tenga ventajas, al tener mejor conexión. También se evitan errores en instantes límites.

Hay muchas estrategias diferentes para realizar el sincronismo, pero en este proyecto se ha escogido el sincronismo en *timebucket*, que se explicará a continuación.

Para un buen sincronismo, se necesita que tanto los dos jugadores como el servidor ejecuten sus acciones en el mismo instante de tiempo. Para ello, se va a obtener el reloj NTP del siguiente servidor NTP: *es.pool.ntp.org*. Se utilizará la clase *NTPUDPClient* que se puede encontrar en la librería de *apache commons net*. Cuando se conecta al servidor NTP se obtiene el *offset*, es decir, la diferencia entre el reloj del sistema interno de cada cliente o servidor y el reloj de servidor NTP.

Por lo tanto, el siguiente código: *System.currentTimeMillis() + offset;* muestra el mismo instante de tiempo en todos los clientes y en el servidor.

Una vez obtenido el mismo instante de tiempo en todos los sistemas, tan sólo queda realizar un *Bucket* dónde se insertará los eventos relacionándolo con un *timestamp*. El *timestamp* se obtiene como se ha mencionado antes: *System.currentTimeMillis() + offset;* y a la hora de añadirlos al *Bucket* se le dará un cierto retraso. El retraso deberá ser ligeramente superior al tiempo que tarda un paquete en llegar a su destino, es decir, a la latencia de la red del cliente al servidor. Se ha considerardo una latencia de 120 ms.

Cada vez que se recibe un paquete se obtiene su *timestamp* y se le añade los 120 ms de retraso antes de enviar el evento al *Bucket.* Cuando se envía un paquete se inserta el *timestamp*, se envía y también se añade al *Bucket* con el mismo retraso de 120 ms. El *Bucket*, es un *PriorityQueue* de la clase *PongEvent* que es la clase que se ha creado para los eventos del juego del Pong.

Finalmente, se procesan los eventos cada 40 ms y en el instante de tiempo del servidor NTP, para que todos se ejecuten al mismo tiempo. Para ello usaremos el siguiente código: *bucket.processRoundEvents(clock.currentTimeMillis());*

Esto permite realizar el sincronismo, si bien hay más formas de hacerlas, se ha optado por el sincronismo por *timebucket*, que es de los más sencillos de implementar.

De manera resumida, consiste en sincronizar todos los relojes y enviar los paquetes con el tiempo sincronizado y añadirlo al *Bucket* con un cierto retraso. Una vez recibido los paquetes con el tiempo sincronizado también se deberán añadir al *Bucket* con el mismo retraso. De esta manera, siempre que el retraso añadido sea superior a la latencia de la red, se ejecutarán los eventos en el mismo instante de tiempo.

3.4 Pong en UDP

3.4.1 Mensajes críticos

El protocolo de transporte UDP no tiene ningún sistema de control de los mensajes, por lo tanto si un mensaje no llega al destinatario, ninguno de los dos se entera. En los juegos en tiempo real, esa característica incluso es buena, ya que no se perderá tiempo en retransmisiones que pueden bloquear el juego. Como se suele enviar información de las posiciones muy seguidas, si se pierde un mensaje, en el siguiente mensaje recibido se actualizará la posición. Pero hay mensajes de estado del juego que no se pueden perder, se ha evitado las pérdidas de mensajes críticos enviando los mensajes cinco veces seguidas. De esta manera si se pierde uno le llegarán los demás mensajes, puede llegar a soportar rachas de pérdidas de hasta cuatro

mensajes. Los mensajes críticos que se enviarán cinco veces son los siguientes:

- PAUSE
- UNPAUSE
- SCORE
- WIN
- EXIT_GAME

Estos mensajes son críticos, ya que por ejemplo al perderse un mensaje de SCORE los clientes no podrán actualizar sus puntuaciones.

Los mensajes MOVE y BALL sólo se enviarán una vez, ya que son mensajes que indican la posición de los objetos y se envían muy seguidos. Si se pierde uno el siguiente actualizará la posición correcta.

3.4.2 Sincronización de IP y puerto

Otro punto que hay que tener en cuenta utilizando el protocolo UDP, es que al no ser un protocolo orientado a conexión, antes de empezar a jugar los clientes deberán de enviar un mensaje de sincronización para que el servidor guarde las IP's y puertos con los que van a jugar. Se ha hecho así porque si no se envían los mensajes de sincronización, el servidor enviaría los datos a IP's nulas o 0. Por lo tanto, hace falta enviar un mensaje de sincronización, y como también es un mensaje crítico se enviará cinco veces. El mensaje se compone de los siguientes campos:

- Mensaje SINCRO:

  - Tipo: 9 (int)
  - Nombre de usuario (String)

El campo tipo se utiliza para que el servidor sepa que es un mensaje de sincronización, y después se envía el nombre de usuario del jugador. Entonces el servidor como sabe los nombres de usuario de los dos jugadores que van a jugar, consulta la IP y el puerto del que proviene el mensaje y lo relaciona con el jugador que tiene el mismo nombre de usuario que se envía en el mensaje. Una vez realizada la sincronización ya se puede jugar con toda normalidad.

3.4.3 Determinación del puerto del servidor

El servidor de juego admite juegos simultáneos, de manera que se puede jugar a más de un juego a la vez. Para ello hay que determinar un puerto distinto para cada juego, para evitar así errores de *socket* en el servidor, porque si ya hay un *socket* UDP abierto, si se abre otro *socket* en el mismo puerto nos dará error.

El servidor sabe exactamente las partidas que están creadas y las va guardando en un vector. Cuando empieza una partida se le pasa al *thread* encargado de gestionar el juego la posición de la partida en el vector. Por lo que si tenemos dos partidas creadas y empieza la segunda partida, que tiene el índice 1, se le pasará al *thread* el número 1. Entonces el servidor determinará su puerto con esta sencilla operación:

PortUDP-npartida

Siendo *PortUDP*=19998 y *npartida* la posición de la partida en el vector, en este ejemplo sería 1.

Para determinar el puerto en el cliente se envía la posición de la partida en el mensaje START. De esta manera, haciendo la misma operación, el cliente sabrá qué puerto está utilizando el servidor para su partida y podrá entonces enviar los mensajes SINCRO para que el servidor sepa qué IP'S y puertos están utilizando los clientes.

3.4.4 Estructura del código en el Cliente

El cliente Android empieza con una actividad, llamada *PlayMultUDP* el *layout* del juego contiene *PongViewMult* que es una clase donde se gestiona todo el juego. Dicha clase, contiene un *thread* (*PongThread*) donde se declara las variables necesarias para el juego y se abren el socket UDP entre otras cosas.

El movimiento de la pala se hace con el *onTouchEvent*, esta función permite saber exactamente qué punto x,y de la pantalla se ha presionado. De este modo, si se presiona dentro de los límites de la pala y se arrastra se puede ir moviendo. Al mismo tiempo, se añade al *Bucket*, el evento del movimiento junto con su *timestamp* y su retraso. Por lo tanto, la pala no se moverá al instante, sino que se esperará el retraso antes de hacer el movimiento. Cuando se añade el evento al *Bucket,* también se envía la posición de la pala al servidor con el mismo *timestamp*. Para realizar los *pauses* y los *unpauses*, se utiliza el mismo método anterior. Si el juego se está ejecutando se enviará un mensaje PAUSE y si el juego ya está en *pause* se enviará un mensaje UNPAUSE.

Para la recepción de los mensajes por parte del servidor se ejecuta un *thread*, llamado *Recibir*, para evitar que se pare la ejecución del *thread* principal. Cada vez que se recibe un mensaje mira el tipo del mensaje para saber los datos que tiene que leer. Una vez leído todos los datos, se añade un evento al *Bucket* con el *timestamp* del mensaje recibido añadiéndole el retraso.

En resumen, se va enviando mensajes a medida que se realiza la acción, pero para la recepción de los mensajes se utiliza un *thread* para no bloquear la aplicación principal.

3.4.5 Estructura del código en el Servidor

La estructura es muy parecida, pero en este caso, el servidor ejecuta un *thread* que va moviendo la bola cada 40 ms, y por lo tanto, también va enviando mensajes de la posición de la bola cada 40 ms.

Para la recepción de los mensajes es igual que en el cliente, ejecuta un *thread* distinto para que no se bloquee el *thread* principal.

3.4.6 Pruebas

Se han hecho pruebas para saber qué porcentajes de pérdidas hay en el juego. Se ha realizado las pruebas sólo con wifi, ya que no se disponía de IP's públicas para realizar las pruebas con 3G.

Se ha podido observar, que si se recibe la señal wifi al 100% no tiene ninguna pérdida y el juego funciona perfecto. Mientras que a partir del 50% de señal, empiezan a haber pérdidas considerables y se observa que el movimiento de la bola no es fluido.

3.5 Pong en TCP

3.5.1 Mensaje críticos

TCP, como su propio nombre indica, es un protocolo de transporte controlado, realiza retransmisiones de mensajes perdidos. Por lo tanto, en el caso de los mensajes críticos, no hará falta que se envíen cinco veces, con una es suficiente. Si no llega el mensaje, el emisor no recibirá el reconocimiento, por lo tanto, enviará otra vez el mensaje hasta que reciba el reconocimiento correspondiente a ese mensaje. Si es el reconocimiento el que se pierde, enviará otra vez el mensaje, por lo tanto, lo recibirá dos veces. Se ha visto con UDP que recibir varios mensajes iguales no era un problema, pues lo mismo pasa con TCP, ya que el código es prácticamente el mismo.

3.5.2 Sincronización de conexiones

El servidor, cuando se inicia el juego, recibe las conexiones de los dos jugadores que van a jugar. Como deberá enviar información a dos jugadores constantemente será necesario disponer de un vector de *sockets* de dos posiciones, la posición 0 será el *socket* del jugador 0 (creador de la partida), y la posición 1 será el *socket* del jugador 1 (el jugador que se unió a la partida).

El servidor, por lo tanto, deberá identificar a quién pertenece cada conexión. Para ello, los jugadores, enviarán un mensaje SINCRO, justo después de la conexión, con su nombre de usuario. El servidor identificará a cada usuario

gracias al nombre de usuario que envía cada jugador, del mismo modo que se ha hecho con UDP.

Como se puede observar, los dos necesitarán una especie de sincronización inicial para el correcto funcionamiento del juego. La diferencia está en que UDP sólo necesitará saber IP y puerto de cada usuario, mientras que TCP sólo necesitará identificar la conexión de cada usuario.

3.5.3 Determinación del puerto del servidor

La determinación del puerto en el servidor se hace de la misma manera que se ha explicado en el Pong UDP, y además como utilizan diferente protocolo de comunicación podrán utilizar los mismos puertos. Es decir, que se pueden tener dos partidas simultáneas de Pong UDP y Pong TCP por el mismo puerto sin ningún problema. En cambio, no se pueden tener partidas simultáneas de Pong TCP y de 3 en raya, ya que ambos juegos utilizan el protocolo de comunicación TCP. Para evitarlo, los puertos utilizados por el juego del 3 en raya tendrán otro rango, suficientemente espaciado para que no haya colisiones.

3.5.4 Estructura del código en el cliente y servidor

Al ser el mismo juego, la única diferencia que hay es el protocolo de transporte, que en lugar de UDP, es en TCP. Por lo tanto, la estructura del código es exactamente la misma.

También dispone del *onTouchEvent*, dónde se pueden mover las palas y enviar la posición. Los *pauses y* los *unpauses* también se realizan en el *onTouchEvent.*

La recepción de los datos también se realiza a través de un *thread* para evitar bloquear el *thread* principal.

Se utiliza el mismo sistema de proceso de eventos con *timestamp* y *Bucket*.

Sólo hay dos diferencias apreciables en el código. La primera es que se utiliza los *sockets* TCP para enviar los datos en lugar de UDP. En TCP, no hace falta poner la IP y puerto del usuario al se quiere enviar los datos, por lo tanto, resulta más cómodo realizar la comunicación en TCP.

La segunda diferencia, es que en el servidor se deben utilizar dos *threads* para recibir los mensajes. Un *thread* se utiliza para la conexión del primer jugador, y el otro para la conexión del segundo jugador. Esto es debido, a que si se espera a la recepción de datos de un usuario se bloquea el *thread* de recibimiento, y no recibe los datos del segundo usuario. Es por eso que se necesitan dos *threads*. En UDP esto no pasa, ya que al no ser orientado a conexión siempre recibe por el mismo *Socket*.

3.5.5 Pruebas

Se ha realizado las pruebas, y la jugabilidad no es la esperada, ya que al llevar poco tiempo jugando el juego empieza a ir mal, y deja de recibir los movimientos de la pelota, entre otros fallos. Habrá que averiguar el porqué de estos errores, pero hasta el momento no se ha tenido el suficiente tiempo para averiguar de dónde provienen estos fallos.

CAPÍTULO 4. JUEGO POR TURNOS. 3 EN RAYA

4.1 Juegos por turnos

Los juegos por turnos siempre se realizan en TCP. No se puede permitir perder un mensaje, ya que seria crucial para el juego y se quedaría parado sin poder seguir.

En estos juegos el servidor es el encargado de dictaminar el turno de los jugadores, y esperar la respuesta de éstos para procesarla y cambiar de turno. Hay varios tipos de juegos por turnos, dependiendo si se permiten turnos simultáneos o alternadamente. El juego del 3 en raya los turnos son ejecutados alternadamente y del tipo posicional, ya que los jugadores empiezan sus turnos en el mismo orden siempre.

4.2 3 en raya

El juego del 3 en raya es de dos jugadores, uno tira las fichas redondas y el otro las cruces, el que haga 3 en raya es el ganador del juego.

Al igual que el Pong, es un juego muy antiguo y famoso, y programarlo no tiene mucha dificultad.

4.2.1 Protocolo de comunicación

Se ha diseñado un protocolo de comunicación sencillo y se intentará enviar el mínimo número de mensajes posibles. En concreto tres mensajes distintos:

- Mensaje TURNO:
  - Tipo: 1
  - Turno jugador (int)
  - Estado juego (int)

- Mensaje MOVE:
  - Tipo: 2
  - jugador (int)
  - posición de la casilla (int)

- Mensaje ORDEN:
  - Tipo 3
  - nombre del jugador (String)

El mensaje TURNO, se envía al inicio de cada turno, y dictamina qué jugador le toca tirar, por lo que la variable del campo turno jugador debe ser 0 o 1.

El campo del estado del juego indica la situación del juego, y puede estar compuesto por tres estados, se indica con un entero:

- Cuando el juego ha finalizado y un jugador ha ganado, se envía en el campo estado un WIN (int 1), y en el campo turno jugador el número del jugador que ha ganado.
- Cuando el juego ha finalizado y han empatado, se envía en el campo estado un DRAW (int 2), y el campo turno jugador es indiferente.
- Cuando el juego quedan más casillas para tirar y nadie ha ganado todavía, se envía en el campo estado un PLAY (int 3), y en el campo turno el número del jugador que le toca tirar.

El mensaje MOVE lo envía el jugador que le toca tirar ese turno, y contendrá su número de jugador. El jugador que ha creado la partida será el jugador 0 y el jugador que se ha unido será el jugador 1. Después, en el campo posición casilla se envía la posición seleccionada por ese jugador. En total hay 9 casillas disponibles que se guardarán en un vector de 9 posiciones, por lo tanto, el campo posición casilla deberá ser un entero del 0 al 8.

Finalmente el mensaje de ORDEN, se utiliza para la sincronización de las conexiones, que se explica en el siguiente apartado.

4.2.2 Sincronización de la conexión

Al igual que el Pong en TCP, hemos diseñado el mismo sistema de sincronización de las conexiones para determinar el usuario que se utilizará en cada posición del vector de *sockets* creado.

4.2.3 Determinar puerto servidor

Para determinar el puerto que utilizará el servidor de manera que se puedan jugar partidas simultáneas, se ha implementado el mismo sistema que en todos los casos. Pero cambiando el puerto de referencia por el 30000, para impedir que un juego del Pong TCP y un juego del 3 en raya, utilicen el mismo puerto y se produjese error de *socket*. Se ha puesto un rango lo bastante distanciado para evitar que lleguen a coincidir los puertos.

4.2.4 Estructura del código en el cliente

Tenemos la actividad donde se gestionan todas las variables del juego y donde se crea un *layout* con nueve imágenes totalmente en gris (casillas vacías):

Fig. 4.1 Juego del 3 en raya

A estas imágenes se le aplica un *OnClickListener* para que se pueda pulsar con el dedo sobre ellas y saber así qué imagen se ha pulsado.

Después se ha creado un *thread* secundario donde se realiza la conexión con el servidor y se entra en un bucle *while (!gameend)* repitiendo lo que hay en su interior hasta que finalice el juego.

El bucle consiste de los siguientes pasos:

1) Se recibe el mensaje TURNO del servidor.

2) Se procesa el mensaje

3) Si el mensaje TURNO contiene el estado PLAY y eres el mismo jugador que contiene el campo turno jugador, entonces es tu turno. Una vez eliges la casilla se envía al servidor y se queda a la escucha para recibir el mensaje MOVE, que en este caso es tu propio movimiento enviado por el servidor. Una vez recibido el mensaje MOVE se vuelve al inicio del bucle.

4) Si el mensaje TURNO contiene el estado PLAY y no eres el jugador que contiene el campo turno jugador, entonces se queda a la espera de recibir el mensaje MOVE del otro jugador. Una vez recibido el mensaje MOVE se vuelve al inicio del bucle.

5) Si el mensaje TURNO contiene el estado WIN, quiere decir que el jugador que contiene el campo turno jugador será el ganador de la partida. Por lo tanto, se cambia a *true* la variable que controla el bucle (*gameend*).

6) Si el mensaje TURNO contiene el estado DRAW, quiere decir que el juego ha finalizado y se cambia a *true* la variable que controla el bucle (*gameend*).

Cada vez que se recibe un mensaje TURNO, se actualiza un *TextView* que muestra el estado del turno. De esta manera, los jugadores sabrán cuando les toca tirar, cuando tienen que esperar al movimiento del otro jugador, cuando han ganado, perdido o empatado.

Cada vez que se recibe un mensaje MOVE, se actualiza la imagen que contiene la posición que se ha enviado por la imagen que le corresponda (redonda para el jugador 0 y cruz para el jugador 1).

4.2.5 Estructura del código en el servidor

La clase que gestiona el juego extiende a *thread*, para que el servidor pueda seguir escuchando conexiones de otros usuarios y creaciones de partidas. Primero se crean y gestionan todas las variables necesarias. La más importante es un vector con nueve posiciones que indicará las posiciones de las casillas.
El servidor al iniciarse el juego se encargará de sincronizar las conexiones de los usuarios para relacionar el primer usuario con el *socket* 0 y el segundo usuario con el *socket* 1.

Después se entra en un bucle, que se repite hasta que el juego finalice. En el interior del bucle contiene los siguientes pasos:

1) Se envia el mensaje TURNO a los dos jugadores con el turno del jugador que le corresponde tirar y el estado PLAY en el caso de que no haya finalizado la partida.

2) Se queda a la espera de recibir el mensaje MOVE del jugador que le corresponde tirar.

3) Cuando recibe el mensaje MOVE de dicho jugador, se añade al vector que gestiona las casillas y se reenvía a los dos jugadores.

4) Antes de enviar el mensaje TURNO comprueba el vector de las casillas del 3 en raya para saber si alguien ha ganado o está todo ocupado.

5) Si ha ganado algún jugador enviará el mensaje TURNO con el estado WIN indicando el jugador que ha ganado, en cambio, si todas las posiciones del vector están ocupadas se enviará el mensaje TURNO con el estado DRAW. En ambos casos finalizará el bucle y por lo tanto la partida. En caso contrario, se vuelve al inicio del bucle.

4.2.6 Pruebas

Se ha realizado las pruebas y el funcionamiento es el correcto, si no se tiene buena conexión simplemente tardará más tiempo en llegar la posición, pero al ser un juego mutlijugador por turnos no tiene mucha trascendencia el aumento de retardo.

CAPÍTULO 5. CONCLUSIONES Y LÍNEAS FUTURAS

5.1 Conclusiones

El objetivo inicial era el diseño de una plataforma multijugador que pudiera disponer de una sala de espera para poder crear partidas y unirse a ellas. Dicho objetivo se ha cumplido, aunque con ciertas dificultades, ya que al principio se diseñó la sala de espera en TCP en lugar de MQTT. En TCP se llegó a un punto en que al crear una partida, no se podía cancelar porque el *socket* estaba a la escucha de una conexión de inicio de partida. Este problema se solucionó implementando MQTT, un protocolo de comunicación muy sencillo y eficaz. Que además permitía añadir la opción de *chat* que en un principio no la se tenía planeada.

En este proyecto he aprendido el protocolo MQTT, un protocolo que antes de realizar el proyecto no sabía que existía. Es un protocolo muy útil para ciertos casos y muy fácil de implementar. Sin duda lo tendré en mente en el futuro para realizar proyectos dónde tenga cabida este protocolo.

He adquirido conocimientos de programación en Android, que está basado en java. Por lo que el aprendizaje no ha sido demasiado difícil debido a que ya tenía conocimientos del lenguaje. Saber programar Android brinda nuevas oportunidades de negocio, ya que Android cada año está creciendo y obteniendo más popularidad y saber programarlo te abre nuevas puertas profesionalmente hablando.

Finalmente, me gustaría decir que he aprendido mucho al realizar este proyecto y actualmente tengo pensado desarrollar aplicaciones Android en un futuro. Ya sea como *Hobby* o en un ámbito profesional, nunca se sabe cuando puedes encontrar una oportunidad de negocio.

5.2 Ambientalización

Se ha tenido en cuenta la ambientalización del proyecto. El proyecto no tiene demasiado impacto en el medio ambiente, únicamente la emisión de señales que se realiza para la comunicación en línea. Dichas señales se han visto reducidas al aplicar MQTT, ya que los mensajes que se envían son más cortos. Se ha desarrollado juegos sencillos donde no requieren mucho cálculo y procesamiento, por lo que las aplicaciones no consumirán mucha batería.

5.3 Líneas futuras

Se han cumplido satisfactoriamente los objetivos iniciales del proyecto, aunque siempre es bueno pensar en el futuro y añadir mejoras para que la experiencia del juego sea mejorada y los usuarios disfruten mucho más a la hora de jugar. Las posibles mejoras que se podrían incorporar son:

- Mejora de la interfaz gráfica. El proyecto no se ha centrado mucho en ese aspecto ya que lo principal es hacerlo funcionar. Una buena interfaz gráfica puede ayudarte a aumentar considerablemente el número de ventas de la aplicación.

- Realizar un sistema de bases de datos para que los jugadores se registren, en lugar de poner un nombre de usuario cualquiera. La base de datos ermitirá contabilizar estadísticas de todos los jugadores y en cada juego de nuestra plataforma.

- Añadir la opción de poder invitar a los jugadores para que se unan a la partida creada. El jugador invitado podrá aceptar o no la invitación.

- Añadir una opción para poder jugar por *bluetooth* sería una mejora interesante, ya que el usuario no se vería necesitado de tener una conexión de datos contratada si lo único que quiere hacer es jugar con sus amigos que también tengan la aplicación instalada.

- Elaborar una librería del proyecto, para que otros programadores que quieran realizar un proyecto de las mismas características lo tengan más fácil.

- Diseñar un sistema de integración de juegos a la plataforma, para que programadores externos realicen sus propios juegos y poderlos implementar en ella.

- Disponer de un servidor con IP pública y subir la aplicación al Android Market.

BIBLIOGRAFIA

[1] Murphy, M.L., *Beginning Android 2*, Apress, USA, 2010.

[2] Hashimi, S., Komatineni, S., Maclean, D., *Pro Android 2*, Apress, USA, 2010.

[3] Silva, V., *Pro Android Games*, Apress, USA, 2009.

[4] Gramlich, N., *Android Programming with Tutorials from the anddev.org-Community*, anddev.org, 2009.

[5] Documentación del protocolo de comunicación MQTT de IBM.
URL:http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-3r1.html.

[6] Lane, D., *Push notifications for mobile apps*, Blog donde describe el funcionamiento de MQTT.
URL: http://dalelane.co.uk/blog/?p=938.

[7] Lane, D., *Using MQTT in Android mobile applications*, Blog donde nos describe como implementar MQTT en Android.
URL: http://dalelane.co.uk/blog/?p=1599.

[8] Android developers web, página web oficial para programadores en Android, puedes encontrar mucha documentación sobre Android.
URL: http://developer.android.com/index.html.

[9] Anddev.org, Android Development Community | Android Tutorials, foro para programadores Android.
URL: http://www.anddev.org/.