A Client Server Model for Multi-Player Network Gaming

Freddie Honohan
B.A.I. Electronic & Computer Engineering
Final Year Project Report, 6th April 2011
Supervisor: Glenn Strong

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Part I

DECLARATION

I declare that the present work has not previously been submitted as an exercise for a degree at any university. It consists entirely of my own work, except where references indicate otherwise. The library of Trinity College, Dublin may lend or copy this thesis on request.

Signed: Freddie Honohan
Part II

ACKNOWLEDGMENTS

First and foremost I would like to thank my supervisor, Glenn Strong, to who I owe a debt of gratitude for his expert help, advice and enthusiasm. He showed me great patience and encouragement and without his guidance this work would never have been completed.

I would like to express special thanks to all the lecturers from the School of Computer Science and Statistics throughout the years, who sculpted my interest in programming and taught me the fundamentals I needed to complete a substantial project such as this one. Lucy Hederman (Computer Science 1), John Waldron and Kenneth Dawson Howe (Computer Science 2), and Hitesh Tewari (Computer Networks).

My gratitude goes also to my fellow Electronic and Computer Engineering students, for the genuine camaraderie within the class, and of course my family for frequent support when it was needed most.

Finally I am extremely grateful to the School of Engineering for producing, and granting me the opportunity to study, such a well rounded course.
Part III

ABSTRACT

Abstract

This project was a practical exercise in programming, logical thought, and an exploration of computer networking and mobile development.

The objective of this study was to explore the various techniques and technologies which are used to provide multiple client networking services. To implement a system for platform independent Client-to-Client networking and to explore the various mobile operating system platforms, with the intention of gaining development experience in one or more.

The study was performed by first looking at all the possible implementations of such a system. Upon full analysis and comprehension of the techniques studied, one was chosen on merit and feasibility, and the development of a system centred around that technique was pursued.

Upon completion, and after lengthy testing of the server, a client model was implemented to interact with it. The client model was designed in order to accurately demonstrate the full capabilities of the developed server.

Additionally, on reflection, thought was put into any additional features which could be added to the server, and structural improvements to produce an augmented communication service.

The project resulted in a significant increase in comprehension of networking protocols, Java programming and a solid foundation for Android application development.
A prototype server model, and a set of example clients for Android were produced.
1 INTRODUCTION

1.1 Personal

I decided to undertake my Final Year Project with the School of Computer Science and Statistics, rather than the School of Electrical and Electronic Engineering. I did this for many reasons, but mostly because of personal preference and interest. The feeling of having code compile and run successfully is something more fulfilling than when one derives an equation correctly. The possibilities of technological advancement are limitless. This astounds me, and as such I chose the most amount of Computer Science credits I could possibly take in Senior Sophister. It seemed fitting, and even more exciting to pursue a Final Year project with the school also.

I decided to try come up with my own project idea, one that would be complex and interesting enough to stand equal, aside those proposed by the school. Merging my interests within a project of my own conception, I felt would benefit the project. It would maintain my concentration throughout, and drive the motivation behind it.

1.2 Project Conception

The rapid increase of smart phone penetration into society was sudden. With such powerful devices becoming so widespread, it seemed a new era of software design was fast approaching. Not only was there going to be a change in user interaction, but also in the purpose of application. The greatly improved hardware capabilities of these phones had cleared the way for a vast range of new possibilities, and would raise the consumer’s expectation of their use.

These were the two main players emerging quickly in the ‘casual’ smart phone manufacturing scene Apple Inc and Google. Research in Motion’s Blackberry and the line of Nokia’s Symbian powered devices were geared towards the business market, and as such, had a decent share overall at that stage because of the “head start” they
received. Businesses worldwide saw these devices as a necessity for round the clock connectivity and organisational purposes.

However, with Google’s Android operating system, and Apple’s iOS geared toward the home user, a shift in the smart phone market share was imminent [Figure 1][1]. The simplification of the UI design, and the ease of access via well implemented touchscreen control indicated that they would prevail over their competitors. Both companies had released their software developer packages, and their respective developer communities were growing. It had become obvious that there was going to be a large market place for mobile software, from simple organisational and management tools to unique media applications and complex three dimensional games.

![Figure 1: Smart phone market share by end of year.](chart)

It seemed only natural to pursue something in the gaming area of computer application. The complexity behind modern game production is intriguing. Exactly how they are structured, and how the different aspects of the design are interlinked were questions I knew
would not be answered quickly or with comprehensive ease. Most game productions these days are multi-million dollar projects, with huge development teams, split up into the various sub departments. Be it animation, gameplay, level design, artificial intelligence or networking, each unique section is interesting in one way or another.

Multi-player gaming has become a strong selling point for computer games. It started with online text based role playing games on Bulletin Board Systems [2], developed into server based First Person Shooter games and then of course to Massively Multiplayer Online Games. Even locally based, single monitor multiplayer adds greatly to the user experience and enjoyment of a game title. However, playing split-screen tactical games is improved immensely with networking. An exploration of the implementation of this kind of service appealed greatly to me, such as investigating the complex infrastructure behind Microsoft’s XBOX-Live or Sony’s Playstation Network, or even the protocol behind the multiplayer games available on N-Gage, implemented wirelessly via Bluetooth.

As this new third generation of phones are expected to be connected to the internet much more than their 2G predecessors. This opens the door to a wide selection of online multiplayer games which could be ported to these mobile operating systems.

1.3 Objectives

The main objective of this project is to explore the techniques surrounding multiplayer network gaming, and to implement a platform independent approach to providing such a service. With initial focus on Android and iOS devices, a communication layer for mobile devices was to be developed.

This report is organised as follows: Firstly the background research of the various methods which could be used to deliver multiple mobile client networking is discussed. In the case of complications with the networking technologies for either operating system, they
too will be outlined. Arguments for and against each potential solution are made, and the reasoning behind the final method chosen is explained.

In Chapter 3, the industry standard protocols for implementing the solution are documented. An implementation of the chosen technique is then discussed, followed by an outline and discussion of the process of designing an Android smartphone application.

The results of the project are analysed in Chapter 4. A critical description of the approach taken is also made, highlighting the improvements which could have been made.

Following this, Chapter 5 deals with the theoretical expansion of this project, outlining the potential additions to the developed system. Recommendations of possible improvements to its features are listed.

Finally a conclusion of the project is made in Chapter 6. A bibliography of all literature for citation, and an appendix of all code of interest is appended to the end of this report.
2 BACKGROUND

Multiple techniques exist to deliver networking solutions for multiplayer gaming or other multiple client applications. These solutions needed to be examined in order to decide which route was to be taken to best develop a cross platform solution to multiplayer mobile gaming. The solutions were to be assessed, concentrating on the feasibility, and also ease of integration for a cross-platform implementation.

2.1 Ad-Hoc Networking (WIFI)

AdHoc Networking is also described as peer to peer networking. It is a wireless technology that allows two or more mobile devices to connect together to form a private network using wireless protocol 802.11 b/g/n [Figure 2]. Type B and G wifi give approximate ranges of roughly 32 meters indoors, and roughly 90 meters outdoors, however type N can give double that [14]. This protocol was first used for military applications, but today usage models include playing games in close proximity to your partners, or exchanging information in a meeting room etc [4].

Two types of 802.11 networks, independent [Figure 2(a)], and infrastructure [Figure 2(b,c)] exist. It is the independent model that delivers the kind of networking needing for quickly creating temporary mesh networks between mobile devices. It is key to note that, in the above figure it says “some stations may not be able to communicate”, but what is really meant is they may not be able to communicate directly.

Each independent device can communicate directly with one another, by way of their unique SSID or Service Set identity. It is by means of the SSID (which is a user determined character string), which each mobile station can differentiate between each other. Each 802.11 network is referred to as a Basic Service Set or BSS. If
two devices connect to the same BSS, they in turn can communicate with each other.

This technology is ideal for close quarters connectivity for multi-player games. Gaming systems such as the Nintendo DS have mesh networking built in, allowing for the exact kind of mobile games networking this project aimed to achieve. However I had my own concerns regarding the capabilities of the mobile devices, or rather, the permissions surrounding the use of such technology.

2.1.1 Complications

It is completely reasonable to assume that most modern wifi enabled devices can support this technology. In fact, they can. But the problem lies with the fact that the smart-phones are 3G enabled. 3rd Generation Mobile Telecommunication standards allow for services including mobile internet. Because of this, service providers of such devices have issues allowing the creation of Ad-hoc networks.
This is because it opens the possibilities of a shared data connection, or “tethering” if you will. Data transfers over tethered mobile phones may violate the terms of use imposed by the respective mobile carriers. In particular it was T-Mobile and their iPhone Data charging saga[5], which was a major deterrent from using the technology.

The problem lies with the potential that they simply could not provide such a service to everyone if they were to allow it. Because of these restrictions, this route was not entirely sensible.

So in an effort to maintain the project within the legal boundaries, and allowing for the potential release of any end product onto the official marketplace an alternative solution needed to be found. (In fact, as of late, Google have introduced native services allowing for such use on certain Android handsets (update 2.2 upwards) [6], as too have some iPhone service providers introduced tethering data plans.)

2.2 Bluetooth Networking

Bluetooth technology was invented by Ericsson in 1994, as a method to replace cable communication [7], by way of forming a short range Ad-Hoc network, consisting of independent wireless nodes that have the ability to dynamically form connections with each other to create a network. Such a network has the abilities to grow, shrink and fragment without having to inform any central infrastructure. It is a technology perfect for implementing Personal Area Networks or PANs.

These PANs are also referred to as piconets. A piconet is a collection of wireless nodes, including a “Master” and one or more “Slaves”. Each node can be either a Master or a Slave, or both, allowing for the integration of piconets to form a scatternet, or a larger network [Figure 3]. Connections must be made via a process known as “pairing” to reduce security vulnerabilities.
Bluetooth operates by transmitting signals between the frequencies 2.402 - 2.480 GHz, and it avoids interference by emitting weak signals $\approx 1$ milliWatt [8]. It does not require line of sight, however the low power limits the range of the Bluetooth device, with the most common power rating (class 2) giving a range of $\approx 10$ meters.

Each master communicates to its slave(s) through a technique called, “spread spectrum frequency hopping”. This allows for any Bluetooth device to connect with up to seven other devices, by means of rapidly alternating the transmission channel frequencies using a pseudorandom sequence known to both transmitter and receiver. This is a key feature of Bluetooth, and would enable multiplayer games supporting up to 8 players.

In comparison with the Client Server Model (which is discussed later), the approaches to implementing a multiple client networking service are slightly different as the Bluetooth network topography allows only the master to initiate connections[9].

Rather than have a Server awaiting connections, one to seven
devices would be set listening for incoming connections. Then one
device runs a discovery of nearby devices, and creates connections
to those devices. From a device point of view, one Client (Master)
creates connections to seven Servers (Slaves). But from the applica-
tion point of view, the Master has the responsibility to forward
the data between its Slaves. Therefore it can be perceived as the
exact opposite, with the Master device acting as the Server, and the
Slaves as its Clients.

Alternatively, one could implement a similar service by setting
one device as a Slave (Server) listening for connections, and have
the Masters (Clients) connect one by one. The “Server” would dis-
connect upon collecting the unique information from each “Client”,
and after each device has connected each device would switch roles,
the “Server” (now Master) would then re-connect with each “Client”
(now Slaves). This approach would take time to initialise, much
longer than the aforementioned setup, but would better resemble
the conventional Client-Server approach.

2.2.1 Benefits

The benefits of using such a technology to implement a network-
ing solution for gaming are obvious. There are numerous advantages
for the consumer. Firstly it could be used for zero cost. Secondly, be-
cause of its low power consumption, the drain on the battery would
be miniscule. For the latter reason, it is a well suited solution for
mobile devices, whose power supplies are of constant concern. Also
devices could be located up to ten meters away from each other,
or even up to a hundred meters apart (if class 1 devices), and in
separate rooms or around corners etc.

2.2.2 Complications

Apple released a proprietary framework for Bluetooth gaming,
GameKit, in 2009. This framework provided libraries for IP con-
nectivity and iOS to iOS Bluetooth networking only. Apple had firmly locked down its implemented Bluetooth stack, and a cross platform solution was the main aim.

With sufficient testing, it could be decoded via packet sniffing or otherwise. If this was to be the case, essentially the project would have proceeded on an alternative route. One of reverse engineering, which is prohibited in the Apple developers agreement. Therefore it was not an option.

Apple pride themselves in the security of their devices and operating systems, and as such had taken careful steps to protecting their iOS devices from Bluetooth hijacking. “BlueJacking” or “BlueSnarfing”, which are processes involving the breach of the “pairing” trust agreement, or without the need for “pairing” to access private data such as text messages, images etc [10]. In order to avoid such major security breaches, Apple have kept their Bluetooth protocol a secret.

To sidestep this obstruction and allow full access of the device features to the developer, iOS devices can be jailbroken. To jailbreak a device means unlocking the full feature set of the operating system, thus allowing developers to run any code on their device, as opposed to only code authorised by Apple themselves. Only recently deemed legal, despite Apple’s lawsuits, jailbreaking a device simply voids its warranty. It allows for users to access alternatives to Apple’s own iTunes marketplace, such as Cydia. Today there are several unofficial file transfer and other applications making full use of Bluetooth for iOS, but these can only be run on “unlocked” devices.

Unfortunately, despite being the method of my preference, the project could not proceed using Bluetooth as its networking option. Taking into consideration that only roughly ten percent of all iPhones are jailbroken, and to maintain the prospect that any potential end products of this project could be distributed in the of-
ficial “App Store”, it was decided that implementing an unauthorised Bluetooth framework would be unfruitful.

2.3 Internet Networking

The third possible technique of delivering a cross platform networking solution for multiple clients investigated was through an internet based approach. As these smart phones are all capable of acquiring internet connections either through the 3G service or using their wifi hardware, this was a definite possibility.

Acquiring a specific mobile devices IP address is an issue however. They are constantly on the move, with their designated addresses changing frequently, and as the user has no control over this, trying to keep a static address is impossible. As such, direct internet communication between mobile clients is unreliable, and it is essentially impossible a satisfactory service.

2.3.1 The Client Server Model

To implement an internet based service for mobile devices, the Client Server Model [Figure 4], which is essentially a message passing system, would have to be used.
A client can connect to a remote host, or server, which provides a service to it, or many clients simultaneously. The clients initiate requests for service, and client server interaction follows. In the case of client to client interaction, each client simply connects to a common Server and it simply forwards the data to and from each client respectively.

To access a server, its host IP address needs to be known and also what port(s) it is listening on. To connect to a listening server a communication socket needs to be created, given the destination port and address. This can be done in various ways, but most commonly is accomplished using one of either two established internet transmission protocols, which are both practiced under the Internet Protocol Suite.

2.3.2 Disadvantages

The Client Server Model suffers a few drawbacks, such as a user cannot play games where there is no Internet infrastructure, or when the connection is too bad, something that would be extremely common for smartphones on a 3G connection. Also, in the case of an
extremely popular server, it could become overloaded and crash under stress.

Furthermore, such an implementation would come at a price. Not only would be consumers making use of the games be charged for the connection. So too, would the developer have setup and maintenance costs associated with implementing such a service. Such as server space rental, and other maintenance specific costs.

2.3.3 Advantages

Despite those drawbacks, however, the Internet based approach had several key aspects which stood out. All Internet capable devices such as Android or iOS devices, already had the infrastructure to make connections to the world wide web. This meant that if the server was designed with any platform in mind, it would be easily capable of allowing them communicate together regardless of their respective operating systems.

Not only did it allow much a greater range for game partnership, on a world wide level rather than in close proximity, but also because it was the most feasible of the three methods it seemed like the best option. So to maintain the cross platform aspect, and because of the unfortunate uncooperative nature of Apple’s Bluetooth protocol, an implementation of the Client Server Model was decided upon despite my initial preferences.
3 IMPLEMENTATION

In order to design such a service, attention needed to be paid to the industry standard networking methods in order to fully comprehend how to implement the system. A general overview of the system follows, with separate sections incorporating attention to detail in both Client and Server design, discussing problems that were encountered and the solutions that were found.

3.1 Internet Protocol

The Internet Protocol Suite is a set of communication protocols which has become the industry standard method of interconnecting hosts, networks and the internet [11]. It is commonly referred to as the TCP/IP suite, for the most commonly used protocols are the Transmission Control Protocol and the Internet Protocol. Within it consists four abstraction layers, the Link, Internet, Transport and Application Layers. For the purposes of this project, focus on the Transport layer was sufficient.

Direct host to host communication is encompassed in the TCP/IP Transport layer, which is a set of the various Transport protocols. The two most commonly used are the Transmission Control Protocol and the User Datagram Protocol for delivering end to end communication services. They can provide functionality such as streaming data, congestion control, reliable data delivery and connection-oriented communication.

3.1.1 Transmission Control Protocol

TCP is the best known protocol, which is why its abbreviated name was added to the abbreviation of the Internet Protocol Suite. Its status is standard, and in practice, most TCP/IP implementations will include TCP. As a connection-oriented protocol, most of the user application protocols, such as Telnet or FTP make use of TCP. It provides a considerable amount of facilities, which combine
into a point-to-point channel for applications that require reliable communications. TCP is one of the most important blocks which the world wide web is built upon.

![TCP Packet Structure](image)

**Figure 5: TCP Packet Structure**

**Advantages**

TCP has many useful features, which provide:

- Guaranteed, ordered data delivery, without any loss or data duplication.
- Connection based transmission, for complete initiation, nego-
tiation and termination management.

- Flow and congestion control for optimal performance.

These features make TCP perfect for data critical applications, allowing no room for error. If a packet is lost or corrupted, it is made known to the sender and resent, and upon reception normal service resumes.

**Disadvantages**

Because of its full connection management, and its reliable data transport infrastructures there is one major disadvantage to using TCP, which is speed.

The extra overhead needed to implement the beneficial aspects of TCP makes transmission of data slower. Appended to a TCP packet [Figure 5], are the sequence number, acknowledgement, and checksum bits among others. This adds a significant amount of data to the packet on top of the actual user defined data that was sent.

3.1.2 User Datagram Protocol

UDP is the second most commonly used transport protocol. Its implementation was intended for modest data transmission applications, or those which can afford to lose a little amount of data (such as multimedia streaming). It is important because, RTP or Real Time transport protocol is supported over UDP.

**Advantages**

The main advantage which UDP has in comparison with TCP, is its speed. As it is a connectionless protocol, there is no connection establishment phase which accelerates the process. A header [Figure 6] is attached to the start of a data packet, containing information
The checksum is a 16 bit number, used to determine whether the data was corrupted in transport. So in this way UDP provides error detection, however will do nothing with the error, other than notify the recipient that the packet is corrupted and unreliable for use.

**Disadvantages**

However UDP cannot facilitate:

- Guaranteed delivery of data.
- Ordered delivery of data.
- Flow control, which is left to the user application.

At the transmission end, UDP continues to send data irrespective of knowing whether it is being delivered or not. Whereas at the receiving end, data can arrive out of order, or not at all. It is for these reasons that UDP is key to providing a service which is as “real time” as possible for time critical applications.
Sockets & Ports

Sockets & Ports are objects “which are needed to determine which local process at a given host actually communicates with which process, at which remote host, using which protocol”[11]. A port is a 16 bit number used to identify which machine process it must deliver incoming messages. TCP uses sockets to establish connections between two end processes. “A socket is a special type of file handle, which is used by a process to request network services from the operating system.”[11]

The Server would need to listen on a specific port, and each client would need to attempt to establish a connection to it by opening a socket with the servers IP address and port. The port number 4444 was chosen arbitrarily.

3.2 Server Design

It was chosen to develop the Server in a language with which previous experience was held. Java was chosen, as it shared many libraries which could be made use of for Android development also. In comparison with C++, of which programming experience was also held, Java performs automatic memory management [12], which is of great benefit to the programmer.

The Java .Net library provides the classes for implementing network applications. Classes included are Server Sockets and Sockets for both TCP and UDP networking. These were the foundation classes for which the Server was built upon. In order to reduce the chance of error, TCP was the transport protocol used initially to implement the game and server service. The main literature used to aid the development of the server was the Java SE language specification [13], and the official online Java documentation and reference.

The Server would need to support both simple and complex interactions. A setup aspect was needed, to communicate with each
client which service they needed, a match-making function to connect or group clients together and then most importantly the game services needed to be implemented. The game services needed to be structured in such a way to incorporate both scalability and reusability, for differences in player numbers. They too needed to deliver functions for simultaneously streaming data between all players, and supply a simpler one player per turn service also.

There were other key features to deliver within the server, in order to deliver satisfactory functionality. Multiple users needed to be accommodated, so too did the ability to service multiple games need to be well implemented. In order for this to be achieved successfully an object oriented approach was taken. With objects implemented for both the clients who connected, but also each game that began.

In the case of the client object, it was designed to store the important information needed to work with it. This included, but is not limited to:

- Its socket object, for communication purposes.
- Its unique identity for management purposes.
- The device type from which the connection was established, (Android, iOS, PC etc.) for statistics.
- The game type it was playing or attempting to play.
- Its playing partner or partners object(s).
- The actual game object that was assigned upon the commencement of play.

The game object on the other hand was implemented so as to provide the system with knowledge of the status of each game. It simply included:
• Its game identification number.
• Its game type and the amount of players playing.
• An array of client objects storing each players details.

3.2.1 Continuity

In order for the server to provide service to multiple clients simultaneously, it needed an architecture that somehow could deal with multiple connections. To provide this ability concurrency needed to be explored. A multithreaded approach was adopted, whereby each connection would spark a new process to deal with the client connecting [Figure 7].

![Multi-Threaded Server](image)

Figure 7: Multi-Threaded Server

At the time of connection, a client object would be instantiated with its unique identification number and the associated socket. The resulting server thread would then proceed processing the rest of the information needed to advance into a game. It was through this simple concept of a multi-threaded server that a basic understanding
of concurrency was gleaned. In order to provide additional game specific services, further multithreading would be needed.

The Server also needed to be able to run around the clock. With the core process simply listening for, and accepting connections, allocating processing space and then returning to listening, most of the infrastructure had been achieved. However it also needed to have a mechanism to erase obsolete objects stored in memory. If the server failed to do this, eventually, after great use the system would begin to run out of memory, weakening performance, and ultimately causing the system to crash.

To implement such a mechanism, a core feature was added. Rather than having a maximum allowed number of connections, achieved by a fixed size array of clients, a linked list of idle clients was put into effect. This allowed the system to store each client upon connection. Upon that client finding a suitable game to play, the server would release the object into memory by removing from the list, allowing the game threads to deal with them, and upon completion of the game, disposing of them completely by mean of the Java Virtual Machine’s automatic disposal features. A function to tidy the linked list was also built, which checked each node for its connection and playing statuses, and was called automatically each time a new connection was made.

3.2.2 Statistics

Several functions were added to keep track of the service being provided. Having a knowledge of statistics such as:

- Who or what devices were using the server?
- How many connections had been established and were currently connected?
- How many games had been and were currently being played?
could give an early indication of the stress of the server. To keep track of the numbers, memory was allocated in the main program to count such information. Functions were implemented to manage the service, such as updateClientStats, start/endGame, and inc/decUpdate.

![Server GUI](image)

**Figure 8: Server GUI**

To display these numbers a simple Graphical User Interface was designed using NetBeans Java Swing GUI builder tool [Figure 9] . Its simple function was to display the counters, and had a scrollable text field which provided information about games beginning and ending.
3.2.3 Game Services

The characterisation of a game was broken into two distinct definitions, turn based play, and continuous play. Turn based play was centred around simple card or board gameplay flow, which are non time critical. Where each player has time to think before acting or taking his or her turn, and each player has one turn per rotation.

Continuous play is based on real time action. It is the service needed to be provided for “real-time” virtual gaming, such as sports, role playing games and first person shooters etc. This is when each player has the ability to act simultaneously, irregardless of their opponents status.

The implementation of the two characterisations needed radically different approaches. The first could be simply implemented with some logical reasoning, and the second would have to require a more concurrent approach. Game class runnables were designed for each approach and for each variation in player numbers.

The service of communication transfer was supplied by the server, with the game logic left up to the clients themselves. Another approach which could have been taken would be that the server would control the game logic, reporting back to the client if they had played an impossible move. However the communication service alone was enough for the server to handle, without having to worry about game logic. Because of this the server needed to be notified when to terminate service, this was put in place by the clients sending a “GameOver” string to the server (an “X” was chosen arbitrarily). This “Handshaking” method was sufficient for a turn based service but needed further adjustment continuous play.

Turn Based Play

To provide a service for turn based play the server needed to simply forward each players turn to its opponents in numerical order. It was decided to implement service for games of two to four players,
this was because it is rare to find board games which support greater than that number. The implementation of a two player game was first undertaken.

![Figure 9: Two player turn based service structure](image)

It was sufficient to implement such a game with only one server thread receiving and forwarding the data [Figure 9]. A two player turn based game was implemented with the class “TwoPlayerGame”. Upon initialisation the class took control of each players socket, input and output streams, to write and read data to and from them.

It began by notifying the clients whether they were playing first or second, and what game ID they were playing in. Then it entered its service loop, waiting for each players input and forwarding it on. Upon reception of the game over string, it would forward the string to the awaiting client and break from the loop. It would terminate by alerting the server that the game was over, updating the statistics, closing each stream it had a handle on, and finally closing the sockets.

A multiple move per turn scenario was also implemented. A multiple move per turn scenario can occur in many games, an simple example of this would be the double jump in a game of checkers. However as can be seen in the class it has been commented out. This is because this scenario would not occur in the game chosen to demonstrate the system.

This was a simple matter of having each client append a chosen end-of-move string to its moves (“EOM”). If the server received a move which did not end with the end-of-move string, it would forward the data on and await another turn from the same player. It
would continue like this until notified that the turn was indeed over then move onto the next player.

Game class runnables were also implemented for three and four player turn based games, nothing changed dramatically from the implementation of the two player game. A handle on more data streams was needed, and the server forwarded input to more than one player instead. The end of game method remained the same, with each player being notified that the game was over and the server thread shutting itself down.

On the clientside, because it is turn based, all the input output can be performed in one process. Problems were encountered with Java input output classes, but were quickly solved upon comprehension of their “blocking nature”. Blocking I/O is encountered when a process is asked to read from a stream, and will not continue until there is something has been put into the stream to be read.

**Continuous Play**

UDP would be the protocol of choice for most continuous multiplayer gaming solutions, because despite the potential packet losses, the received data would be the most up to date. This is key to deliver a “real time” aspect to the game. However as the initial connection and turn based solution had been implemented using TCP, it was decided to continue working with it to develop the architecture needed to produce a continuous communication service.

As with the turn based implementation, design began with the implementation of a two player model. Separate processes for both input and output stream needed to be introduced at the client end, whereas the server needed a thread per player to forward its sent packets [Figure 10].
Within the server, game class runnables were written to provide the service for up to four players (Two/Three/FourPlayerContinuousGame), in the above figure a two player runnable is highlighted in the area bounded by the dotted line. Each of those sent the clients their player number (though irrelevant) and game number, they then instantiated and ran a PlayerToPlayer(s) thread per client, and joined each of those processes.

To “join” a process is essentially the action of pausing the process until the internally invoked thread has completed. This enabled the game process to wait until handshaking had occurred and both (or all) of the sub communication threads had completed before shutting itself down and alerting the main server process of the disconnects.

The main complication of this design came when a handshaking method needed to be devised. Instead of simply sending an “X”, an acknowledgement from the recipient was needed. It was necessary for this to occur, as it is impossible to shut down a thread remotely if they are in a blocking state, waiting to read or write to a stream. This would serve to notify each of the six active threads chronologically, allowing for a clean and successful end to the service.
The theory was straight forward:

1. Client 1 sends the “X”, ending its output process.

2. The player 1 to player 2 communication thread receives and forwards the string then ends its service.

3. Client 2 receives the notification at its input thread, which remotely ends its output thread, takes a handle on the sockets output stream itself, sending an “XX” as an acknowledgment and ends its service.

4. The player 1 to player 2 communication thread receives and forwards the acknowledgement then ends its service.

5. Client 1 receives the “XX” at its input thread, and shuts down content.

However much frustration was caused upon testing, for a socket was being closed mysteriously, leaving the server to crash as one of its threads was left in a reading state.

In the article “Reading from and Writing to a Socket” in the Java documentation it says after some example code to close a socket - “A well-behaved program always cleans up after itself, and this program is well-behaved. These statements close the readers and writers connected to the socket and to the standard input stream, and close the socket connection to the server. The order here is important. You should close any streams connected to a socket before you close the socket itself.”

This implied that to close a socket successfully the underlying streams needed to be closed also. However after many stressful hours spent debugging, it was found that upon closing an input stream of a socket, the socket itself would close. This was rectified and the “handshake” procedure was successfully completed.
3.2.4 Match Making

Upon connection, the client was dealt with by a unique server thread in order to establish which game it intended to play. To access the different types of game, the client sent an integer value corresponding to its chosen gametype. 1-3 where used to select a two, three or four player turn based game service, and 4-6 where used to select a two, three or four player continuous play game service. When known, the server needed a utility to find a suitable client object which was also idle waiting for a partner. In order for this to happen, the linked list was utilised.

Methods were written (findPartner and findPartners), in order to attempt to find a suitable partner or suitable partners to initialise a game. The methods simply checked the list for idle clients awaiting partnering for the chosen game type.

Each clients server thread used the appropriate function once, and if partner(s) were found, the client objects were marked “paired”, given pointers to their respective playing partner(s), and the game was initialised. However if there were no suitable partners, or not enough, the thread died, leaving the client object idle in the list, and the client itself awaiting notification of successful partnership.

Synchronised access needed to be used in order to ensure that no collisions of access to the list occurred, which would have resulted in errors. The keyword “synchronized” is a locking mechanism within Java, used so that no two invocations of the same method may interleave, essentially taking exclusive access of the variables until completion. The errors caused could be significant, for example:

A client is idle in the list, waiting for someone to play a two player game. Two clients connect simultaneously, and both try accessing the list to find a second player and find the idle client and attempt to play. Both attempts at partnership are successful, however actually someone doesn’t get a partner, worse still the idle client now has

34
two!

All sorts of complications would occur. To leave the "available
partners" list volatile like that would have been disastrous.

With the server designed and tested using a simple java console
based test client, a mobile client was needed to demonstrate its capa-
bilities. As the server was built around established transport proto-
cols, and communication was simply dependent on the transmission
of strings, the necessary structure for cross platform compatibility
had been accomplished. It was decided to focus on delivering a com-
plete Android application, rather than attempting to quickly throw
one together for iOS also, because of time constraints.

3.3 Overview of Android Client

Android is an operating system designed specifically for mobile
devices. Google purchased the start up, Android inc, who originally
designed it in August 2005 [15]. They built upon the stack and
released the first Android SDK in November 2007. The SDK pro-
vided the tools and APIs necessary for developing applications on
the Android platform. Java is the programming language used by
the system, however a native development kit or NDK has also been
released allowing developers the ability to program certain parts of
an Android application in native C or C++.

The features of Android which the SDK includes libraries and
references to include:

1. Optimised Graphics with a custom 2D and an OpenGL ES 1.0
   based 3D engine.

2. Media support, for common audio, video, and still image data
   formats.

3. Hardware dependent networking utilities, 3G, Wifi and Blue-
tooth.
4. Other hardware dependent features, such as cameras, GPS and accelerometers etc.

It also comes with an excellent development environment, including a plugin for the Eclipse Java IDE and device emulators etc. It truly is a well developed SDK, with plenty of articles describing the features and how to implement them.

Consequently, and since the server is programmed in Java, the development of an Android client was chosen over an iOS client, avoiding the problems which may have arose when switching to an Objective-C based development kit. The implementation of the client was quick, and remarkably straightforward since the Java .Net library could be used.

To demonstrate the operation of the server, both a simple turn based game and a continuous style game needed to be implemented on Android. A “Snakes’N’Ladders” game was implemented, and another simple 2D exercise was devised to demonstrate the continuous play facet of the server. Several core aspects of Android needed to be understood however, before the application development could begin.

The main literature used to aid the development of the Android application was the official online Android developer guide and reference [16].

3.3.1 Application Fundamentals

The toolset compiles Android applications into an archive file with a .apk suffix. Both the code and all the resource files gets packaged into this apk file, and it is this file which the android-powered devices use to install the application.
In order to avoid issues with the fragmentation of android, libraries which have been marked “deprecated” should be avoided. If a broad target of all versions is intended, the use of newly added libraries to the most recent version should also be avoided. The fragmentation of Android has come about by the rapid release of several versions [Figure 11].

Because of the open source nature of android and because it can be used commercially for no charge, many hardware developers have released Android smart phones or tablet PCs. Similar to the fragmentation issue, targeting the broad range of devices is difficult. As they all have varying hardware configurations such as screen size, and the inclusion or exclusion of certain components (gyrosopes, accelerometers etc).

**Components**

There are four types of application components, activities, services, content providers and broadcast receivers which are the building blocks of an android application.
1. A service is a component which runs in the background, such as playing audio while using other features of the device.

2. A content provider manages the infrastructure needed to access stored data for an application, be it stored locally or online.

3. A broadcast receiver is a component which responds to broadcast announcements. The System itself broadcasts wide announcements, and applications in turn can also broadcast announcements. This would be used for creating status bar notifications or intercepting text messages etc.

4. An activity represents a user interface, it controls the screen layout, and is the only component utilised in this client application.

To activate a component, an asynchronous message named an “Intent” is used. These define the action to perform the task, requesting the action of the specified activity, service or broadcast receiver (content providers are started differently).

The class Activity is the core of an Android application, each application can have one or more. An activity has a lifecycle, which is interrupted or paused when an intent begins a new activity. The activity lifecycle is displayed in Figure 12.
Figure 12: The Activity Life-Cycle

The entire activity life cycle is between the `onCreate()` where the UI is constructed and resources are acquired and `onDestroy()` method where everything is released from memory. The visible life time of the activity is between `onStart()` and `onStop()`. Between the two methods the state of the activity should be saved so that if another one comes to the foreground then comes back to the original activity you find the state the way it was left. The foreground lifetime is between the `onResume()` and `onPause()`. During this time the activity is fully interactive with the user.
Resources

An Android application is made up of one or more components. Each of these components must be declared in the manifest file, which also declares all the application requirements such as minimum android version and any hardware configurations needed by the application.

The manifest file is defined in XML (Extensible Markup Language), which is a data format for structurally organising information. To request the use of hardware components, permissions are declared within the file. These are a set of pre-defined strings which, if included, enable the application access to their corresponding hardware component or service such as internet access or reading the accelerometer values.

As the manifest file allows access to hardware features of the device, there is an automatically generated resource file allowing access to media resources which are used by the application too. Resources are kept in the 'res/ ' directory of the application, which groups them together by type and configuration. In the case of targeting multiple device configurations for example, images need to be created for screens of varying sizes, storing them in the high to low resolution folders respectively.

Sound effects and layouts would also be located in this folder and at runtime a “R.java” file is compiled, storing pointers to each resource in the resource folder.

User Interface & Interaction

A blank UI layout file is created upon creation of a new project, and like the manifest file it is organised in XML. The UI elements can either be declared in this XML file, or can be instantiated programmatically at runtime. There is a free of charge XML creator program specifically for Android named DroidDraw, this gives the user a graphical representation of a screen, and they can drag and
drop UI elements onto it, generating the XML file when complete. The advantage of using the XML file to declare the layout is that it separates the presentation of the application from the code that controls its behaviour.

Layouts can have multiple pre-designed elements within them, such as buttons textViews etc. Or if the user prefers, they can over-ride certain features of those classes and create custom elements. Each element is defined with a unique attribute 'id', and further attributes are used to define the size, colour and many other aspects of the view object. To access a layout object, typically in the onCreate() method of the activity an instance of the object is created and linked to the layout file like so:

```java
Button myButton = (Button) findViewById(R.id.my_button);
```

An elements class functions, such as onClick(), can then be over-ridden and used to handle UI events.

A layout is defined as linear, table, relative or absolute. For this application a linear layout was used for every activity. A linear layout aligns all of its objects in one direction, be it horizontally, vertically or both depending on how its orientation attribute is defined. It allows one element per row, and also the ability to assign a weight to its children. Allocating weights essentially assigns an importance to a UI object, e.g: Three buttons, of weights (2, 1, 1) would produce a layout which the first button would take up twice the amount of the others [Figure 13].
In order for the layout design to fully utilise the entire screen, UI objects can be assigned attributes to “fill_parent” or “wrap_content”. These attributes (if set for width) would fill the entire width of the layout, or be as wide as its content (such as button text) respectively.

To notify the user of events which occur in the application, there are several options possible with Android. Toast, dialog and status bar notifications are documented fully in the SDK. For the client application, there was no need for status bar notifications, toast and dialog notifications were used when required [Figure 15].

Figure 13: Equally weighted “fill_parent” button UI
A toast notification, best for short text based messages, pops up on the surface of the window. It fades in and out for the time specified, and does not handle any user interaction. A toast notification was used in this application to inform the user of its success in attempting to connect to the server, and also to display dice rolls for both its users, and its opponents upon reception.

A dialog notification is a small window that appears in front of the active activity. The dialog takes control of all UI, and are normally used to display loading bars or short messages which require confirmation from the user. For this application, a dialog notification was used to show the user that the application was waiting for a playing partner or partners to be found.

A small problem occurred when implementing this feature, which was causing the application to crash on one of the devices. As the client gets informed of partnership success by receiving its player and game number, which would close the dialog, it was found that this was the cause. The solution was easily implemented. By delaying the server sending that information, the second client application would have the chance to display the dialog for a second. Rather than immediately being told to close it, which was causing the crash.
3.3.2 Game Design

Instead of describing the implementation of both games in sequence, the outline of the game designs will be discussed simultaneously and comparatively. Firstly the application needed a UI interface to allow the user choose which games it wanted to play. This was achieved by creating the start screen, with two buttons to choose either Snakes’N’Ladders or the Continuous Demo. Then each game needed a selection UI to determine how many players it wishes to compete against. However in the case of the Snakes’N’Ladders game, a two player game was the only version implemented. Whereas the continuous play supported two to four players inclusive.

Each of these screens had their own layout files defined, with each button receiving equal weight and defined to wrap content vertically, and to fill the screen horizontally. Each of these buttons were linked to intents which launched the chosen activities. An attractive light blue colour scheme was also implemented.

For both games, a request for disabling the window title was included in order to maximise the screen size. The title bar simply informs the user of the name of the application, which wasn’t needed. For the continuous play a feature request for fullscreen was also added.

Next came the implementation of the actual games. For the Snakes’N’Ladders game, a simple game thread was initialised from the onCreate() method, which dealt with all of the UI, game logic and networking. A class Panel was created which took care of the board and piece animations [Figure 15].
For the continuous play demonstrative aspect of the application a gameThread was also set up. This initially took care of the game setup and partnering, then it started two other threads. One thread for outputting data across the network, the other to take care of animation.

3.3.3 Logic & Control

For the Snakes’N’Ladders game an OnTouchListener of the entire screen was implemented to allow the user to roll a dice. This enabled the application to intercept this event and enabled it to output the randomly generated number on the socket for sending. A locking mechanism was implemented, in order to disable this feature when it was the opponents turn. This was essential to create a good user experience, as the screen is constantly being touched, and if it had not been implemented the dice rolls would be added to a queue each time the screen was touched, and seem to be generated
In order to implement the game logic, a class Disc and an array of board positions was implemented. A function checkRestingPoint() was written, and it was called at the end of a turn to check whether or not the disc in question was on a snake or a ladder. If it was, the discs row, column and position coordinates were adjusted to the end of the snake or the top of the ladder for each case. A checkIfWon() function was also written, which returned a boolean value, and it too was called at the end of each turn.

By using the disc class it was relatively simple to implement the movement of the pieces, which is not linear. Checking the row coordinate of the piece, it was easy to determine which direction the piece should move, as it alternates between left and right each row.

For the continuous play demonstrative aspect of the client, it was decided to mimic a virtual world where each player could move independently and simultaneously. This was achieved in two dimensions, with each player receiving a coloured circle (or “ball”) on a black background. There was no aim to this “game”, other than to demonstrate the capability of the server [Figure 16].

To move their ball, a tilt sensor listener was implemented, allowing the user to “roll” their balls around the screen. The orientation sensor, which is a direct child of the accelerometer, was used in order to intercept pitch and roll readings. These readings allowed the addition of a mock acceleration element to the application, where each ball would be moved by a factor of the pitch and roll.

An updatePhysics() method was used to control the motion of the ball. This took into account the sensor values, and displaced the balls from there previous points accordingly.
3.3.4 Animation and sound

For the Snakes’N’Ladders game sound effects were introduced, in order to improve the user experience. This was achieved by the use of the MediaPlayer class. In each invocation of this class it was necessary to create a player with the specified sound file, set its looping attribute to false, start the the playing sound effect, make the thread sleep until it had finished and finally release the player from memory. Sound effects were implemented for each dice roll, each disc move, and in the event of landing on a snake or a ladder.

To animation process for this game required some thought, and it was decided to implement the disc class as previously mentioned. A move function was implemented alongside the other disc functions,
which moved the piece forward one position at a time. Upon the roll of a dice, or reception of the opponents roll, the disc.move() function was called in a for loop upper bounded by the amount rolled. Each single move distance was calculated by a percentage of screen width. They were separated by putting the thread to sleep between each increment, and the Panel containing the board and piece Bitmaps was redrawn.

Bitmaps were generated and inserted into the appropriate resource folder, for each screen size (low, medium, and high density). A large snakes and ladders board BMP image was used and resized for each resolution, along with two blue and yellow disc PNG images. The PNG image file format includes transparency, Android can support this, and it is for this reason that they were used.

No sound effects were used for the continuous demo. The animation was handled in a separate thread to the game logic. This thread was programmed to redraw the game every 30 milliseconds. A class Panel was implemented for this game, like the Snakes’N’Ladders game. Its onDraw() method called a method doDraw() which took care of the graphics.

As the game began each ball was placed in the centre of the screen. Each users ball was drawn last, on top of the others, so that they could always see it. This was implemented in a quick case statement, which took into account the players number. Blue, green, red and yellow balls were used to draw players 1-4 respectively, with their coordinates dependent on the updatePhysics() method or the received data.

In order to restrict the mobility of the balls, boundaries were introduced so they could not leave the screen. This was implemented in the updatePhysics() method, and each client had to only take care of their own balls location. The radius of the balls was calculated as a percentage of the devices screen width, in order to keep everything...
to scale between players.

3.3.5 Networking

The Java .Net library was used to implement networking for this application. Extensive experience of this library was gained from the implementation of the server, because of this the implementation was similar to the console based java test client.

For the Snakes’N’Ladders game, each dice roll integer value was simply sent over the network and the opponents dice roll was received. This was implemented in a loop until someone had won, in which case the loser would notify the winner by sending the end-of-game string. When this occurred, each client disconnected, an intent brought them back to the menu screen activity and the server terminated its service.

The structure for the continuous play was significantly different as mentioned before. Each client needed two threads, one for sending and one for receiving the coordinates of the balls. The sending was taken care of by an outputThread(), which simply put the coordinates of the ball into the sockets output stream if it had changed location since the last packet of data sent.

The reception of opponents coordinates was implemented in the main game thread. It decoded the data, and changed the variables of the respective ball accordingly. The structure of the data string was designed as follows:

“Player Number : Screens height %: Screen width %”

It was essential to include the player number in order for the recipients to determine which players ball coordinates the values were associated with.

In order for the correct scale to be achieved across a wide range of devices with various screen sizes the coordinates of the balls themselves could not be sent. This would have resulted in a limited range of ball location on a tablets screen, if the opponent was using
a smart phone with a smaller screen size. So the data to be sent was calculated by \(( \text{balls x position} / \text{screen width})\) and \(( \text{balls y position} / \text{screen height})\). These percentages were sent as doubles, and were re-calculated at the recipients end by multiplying by the screen dimensions.

To complete the demonstration the end-of-game string was sent upon a player pressing their “back” button. When this occurred, the handshaking process began. In order for the server to accommodate multiple players (greater than 2), a counter was implemented to count each acknowledgment, when complete it then knew to end service.

3.4 A UDP Model

Nearing the date of presentation of this project a decision was made to explore the possible implementation of a UDP model for continuous play, rather than improving the Android client aesthetically or otherwise.

The architecture of the service needed to be provided by the server, and the client communications were in place using TCP as its protocol. To implement a UDP service, this architecture could be used. The Java reference was researched, and the DatagramSockets were found as the means to use UDP as a communication layer for Java.

The difference between the implementation of TCP and UDP in Java are outlined as follows:

1. The data sent via UDP needs to be sent as a packet containing a byte array.

2. Rather than opening a socket to a specific IP address and port, the Datagram socket simply opens a port.

3. The Datagram packets themselves are specifically addressed to the hopeful recipients IP.
4. There is no specific socket intended for server use for UDP.

Implementing a UDP layer within the server involved a process of TCP initialisation. This included collecting the IP addresses of both clients. The Server then opened a datagram socket on a chosen port, and upon reception of packets, checked the senders IP address, and forwarded it onto the other hopeful receivers IP address. A alternative Android client was implemented with UDP, its purpose remained the same however, simply to send the balls coordinates. A Java console based client was implemented to receive and display UDP in action. Additional variables were added to the Client object class such as lastPacketNo, an InetAddress and a string representation of the IP address for UDP connection management purposes.

To take advantage of UDP for continuous gaming, there needed to be a structure in place for flow control. To enable the dropping of late packets, a packet numbering system was devised. Each packet was numbered from 0-1000 reverting to 0 when 1000 was reached. As the server received packets it checked their number against the last packet number received from the same client, if the number was greater it forwarded it on and saved the packet number as the last. If it wasn’t the most up to date however, it did nothing with it.

In order to manage the packet number wrapping around back to zero, a separate condition was added which checked if the new packet number was less than 100, and if the last packet number was greater than 900 forward it on and save the packet number. With the large packet number conditions in place (100), the system would only fail if no packets were received between 0-99, and 901-1000 in one loop. If this did occur however an entire 1000 packets or more would be lost.

This packet checking was performed at both the intermediate server, and at both client endpoints, providing a dual service. Each Datagram packet was given a size of 128 bytes, and the important
data was separated by a ‘:’, this was needed because the byte arrays were not being byte stuffed, and contained irrelevant information.

This implementation worked well, however the end-of-game string method for terminating service could not be relied upon, because of the strings potential loss. However it provided much cause for thought.
4 RESULTS

This project resulted in the production of a robust prototype server for multiplayer games and a set of Android client models. The architecture needed to provide a service for connecting multiple clients was found using a client server model. An increased comprehension of TCP/IP networking transport protocols was obtained, along with considerable experience working with both TCP and UDP.

To provide the continuous play service, not too dissimilar to most modern multiplayer client server models, a foundation for learning and understanding concurrency was established. Along with a significant body of object oriented java programming work.

The exploration of the Android operating system resulted in a good basic knowledge of its fundamentals for further development on the platform. The two games implemented also allowed further comprehension of the structural aspects needed to implement a computer game on any platform.

The UDP implementation raised issues regarding the compatibility of such a service with mobile devices connected to the internet on 3G networks. Because of the need for the IP address of the clients, and the fact that each of these devices would have dynamically allocated addresses, the feasibility of providing such continuous play for users on the move has not been addressed.

The project suffered as a whole because of initial apprehension regarding the feasibility of using Bluetooth as a protocol. The time spent researching this could have been used beneficially. Further delays came with confusion regarding the operation of a sockets input and output streams, and a hospital visit. Time spent implementing a server GUI could have been better used also.

The project could have met its desired aims, an implementation of an iOS client alongside one for Android, had its complete set of
goals been fully established at the outset. The implementation of a continuous service decided late, was felt to be more fruitful than a stab at learning how to program with the Objective-C language for iOS. I certainly believe, despite my wish to gain iOS development experience, that this project has left me in a more informed position with regards software engineering.
5 FURTHER WORK

There are many beneficial features which could be added to the server, improving this project as a whole. The first is a logon requirement, which would involve the transmission and storage of a username. This could be achieved by initially asking the user for a username, then storing it locally on the device for further connections. Each username could be attached to a client object, and a separate linked list could be introduced listing each and every user. An SQL database stored online is an alternative solution to this problem, Android users could access the database using a content provider perhaps.

With such a dataset in place, users could be allowed the opportunity to choose their partners. It also would be essential to implement some kind of disconnection management. If a client disconnected the game could be paused, or not, and implementing reconnection support to the game would be greatly aided by the database.

A simple addition to the project would also be the implementation of a boolean based switch, allowing the server to turn on and off its multiple move per turn services as it sees fit. So too, would the inclusion of this service into the three and four player turn based game classes.

With regards the continuous play aspect of the server there is one small addition, and a large restructuring which could occur. Support for larger numbers of players could be implemented rapidly with ease. This could be done by expanding the PlayerToPlayers class, and adding more continuous game class runnables.

The major restructuring idea came about during the exploration of UDP near the end of the project. The problem which arose when attempting to implement the termination of the service, could be resolved by combining the two protocols. Using TCP for controlling setup, and critical data flow such as ending the game, and using
UDP as an underlying communication layer [Figure 17]. This could be achieved by implementing additional processes for each stream, and for each transport layer.

![TCP & UDP service structure](image)

Figure 17: TCP & UDP service structure

In fact, as the server would listen on a single port for the UDP layer, potentially the system could work by servicing all the games UDP communication from a single thread [Figure 18]. This thread could forward packets of data from a client to its partner(s), making use of the clients game object to determine exactly who they are.
It is highly likely that this could cause a bottle neck junction, so to avoid the problems of congestion, a global Datagram socket could be declared at the parent server process and a thread fired one for each of clients connected from the game runnables. These would not accommodate a specific client, but rather serve to spread the computation and service across multiple processes. Figure 18 displays the same set of games and partners as in Figure 17, however with a distribution of communication services across multiple threads.

Furthermore, this project was tested on a local wifi network only, which normally generates a latency of less than ten milliseconds.
Latency testing should be performed across the Internet to discover the delays users would actually be subjected to. This could be performed by using a service such as DYNDNS, which would enable the server to be constantly accessible at the same address. Testing for both wifi, and 3G based connections could be carried out.
6 CONCLUSION

The objective of this project was to explore the different solutions of providing multiple client networking which are in use by the industry today, and to develop a solution for cross platform networking between Android and iOS devices. An internet based approach was decided upon, as both mobile operating systems included support for TCP/IP networking.

The Client-Server model enabled the cross platform communication desired, and the byproduct of implementing such a system contributed significantly to the comprehension of concurrency and networking. The improved structural design, incorporating both transport layers, which the project theoretically outlines would not be too dissimilar to techniques used in the network gaming industry today.

In a sense, this solution was the culmination of the project, despite how obvious it seems upon reflection. The deep insight into the networking area of computer application gained from this project allowed me to arrive at this conclusion independently.

An Android application was developed to interact with the server, and the endeavour resulted in valuable experience working with the operating system.

Android has now acquired more than a third of the US market making it the largest single operating system in the United States [17]. However those statistics do not include the Symbian operating system, which was considered to be a targetted platform at the early stages of this project. It was likely overseen in those statistics as Nokia announced a merger with Microsoft in February earlier this year [3]. They were to adopt Windows Mobile as their primary smartphone OS, effectively dumping Symbian.

The original intentions of the project centred around the use of bluetooth to bridge the gap between iOS and Android. Despite this
impossibility, the approach taken has given a solid concept of how to implement such a service, via bluetooth on Android. The following excerpt indicates why this would be beneficial:

“Bluetooth’s performance characteristics better suit multiplayer gaming than slower, intermittent wide-area connections over TCP/IP. Most important, message latency is much smaller with Bluetooth, ranging from 40 to 400 milliseconds on average, and degrades in proportion to the size and number of messages transmitted on the network. This exceeds on average the best performance of TCP/IP on the newly deployed 3G wireless networks. Bluetooth connections are also more robust in the face of interference, and dropped connections due to loss of signal are rare.”[18]

The Client Server model developed has resulted in the procurement of a wide range of technical know-how, previously alien. An implementation of a Bluetooth client server model (as discussed in section 2.2), could be performed by creating an adhoc network between a scatternet of clients. The master would adopt the role of the server after complete discovery of the PAN, it too could also act as a slave in parallel. It would then reconnect each slave to the server, including its own client process.

Readers of this project report could get a glimpse of the logical steps needed to produce a multi-threaded server capable of hosting multiple client communications. They too would be introduced to the Android operating system, and gain development knowledge from it.

This was a thoroughly enjoyable project, which delved into previously unencountered aspects of computer application.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smart phone market share by end of year</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Ad-Hoc networking between multiple devices</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Bluetooth Scatternet Diagram</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>The Client-Server Model</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>TCP Packet Structure</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>UDP Header Structure</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Multi-Threaded Server</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Server GUI</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>Two player turn based service structure</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>Two player continuous play service structure</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>Android OS Version distribution, February 2011</td>
<td>37</td>
</tr>
<tr>
<td>12</td>
<td>The Activity Life-Cycle</td>
<td>39</td>
</tr>
<tr>
<td>13</td>
<td>Equally weighted “fill_parent” button UI</td>
<td>42</td>
</tr>
<tr>
<td>14</td>
<td>Toast &amp; Dialog Notifications</td>
<td>43</td>
</tr>
<tr>
<td>15</td>
<td>Two player “Snakes’N’Ladders” Game</td>
<td>45</td>
</tr>
<tr>
<td>16</td>
<td>Four player continuous play demonstrative “Game” on Emulator</td>
<td>47</td>
</tr>
<tr>
<td>17</td>
<td>TCP &amp; UDP service structure</td>
<td>56</td>
</tr>
<tr>
<td>18</td>
<td>Single UDP communication thread, servicing multiple games</td>
<td>57</td>
</tr>
<tr>
<td>19</td>
<td>Multiple distributed threads for UDP communication</td>
<td>57</td>
</tr>
</tbody>
</table>
References

[1] Gartner Inc. (Gartner.com) - February 2011
[8] How Bluetooth Works, Curt Franklin
[10] Intro To Bluesnarfing, Williamc and Twinvega
[12] Memory Management in the Java HotSpot™ Virtual Machine
[14] 802.11n Delivers Better Range, Phil Belanger and Ken Biba

Appendix

The following Java classes, which were written during this project, are included in this order:

Server Classes:

1. Client Object
2. Game Object
3. Server
4. ServerThread
5. TwoPlayerGame
6. FourPlayerContinuousGame
7. PlayerToPlayers
8. UDPGame

Android Files:

1. Manifest
2. Continuous player amount selection xml layout
3. Two player Snakes’N’Ladders (SNLGame) Activity
4. Four Player Continuous demo (Ball4) Activity

The rest of the 36 classes or activities are included in the CD. Along with the project files, and resources etc. Most importantly the system can be tested locally with the Connect.java runnable.

(The IP address of the connection needs to be changed to that of the host machine)
JAVA CLASSES

/////////CLIENT CLASS///////////
package listserverwithgameno;
/**
 * @author freddiehonohan
 */
import java.net.InetAddress;
import java.net.Socket;
public class Client {
    protected Socket socket;
    public InetAddress ipAdd;
    protected int ID; // Client Identity
    protected int clientType; // 1:Java test, 2:Android, 3:iPhone
    protected boolean playing; // Client status
    protected int playerNumber;
    protected int gameNo; //
    protected Client partner; // Game partner
    protected Client partners[]; // Game partner(s)
    protected int gameType; //
    protected int noOfGamePlayers; //
    protected Game game;
    int UDPPort = 4444;
    int lastPacket = 0;
    String stringIP;
    public Client(Socket sock, int id) {
        playing = false;
        socket = sock;
        ID = id;
    }
    public void setPartner(Client c) {
        partner = c; // Game partner
        playing = true;
    }
    public void setPartners(Client[] p, int players) {
        partners = new Client[players];
        for (int i = 0; i < players; i++) {
            partners[i] = p[i];
        }
        playing = true;
    }
}

/////////GAME CLASS///////////
package listserverwithgameno;
/**
 * @author freddiehonohan
 */
public class Game {
    protected int gameType; //
    protected int noOfGamePlayers; //
    protected int gameNo; //
    protected Client players[];
    public Game(Client client, Client client1, int gN, int gT) {
        gameNo = client.gameNo = client1.gameNo = gN;
        client.playerNumber = 1; // This Client goes first
        client1.playerNumber = 2; // Partner Client goes secondgameNo=client.gameNo=client1.gameNo=gN;
        noOfGamePlayers = 2;
        players = new Client[2];
        players[0] = client;
        players[1] = client1;
    }
}
Game(Client client, Client client1, Client client2, int gN, int gT) {
    gameNo = client.gameNo = client1.gameNo = client2.gameNo = gN;
    client.playerNumber = 1; // Set playing numbers
    client.partners[0].playerNumber = 2;
    client.partners[1].playerNumber = 3;
    noOfGamePlayers = 3;
    gameType = gT;
    players = new Client[3];
    players[0] = client;
    players[1] = client1;
    players[2] = client2;
}

Game(Client client, Client client1, Client client2, Client client3, int gN, int gT) {
    gameNo = client.gameNo = client1.gameNo = client2.gameNo = client3.gameNo = gN;
    client.playerNumber = 1; // Set playing numbers
    client1.playerNumber = 2;
    client2.playerNumber = 3;
    client3.playerNumber = 4;
    noOfGamePlayers = 4;
    gameType = gT;
    players = new Client[4];
    players[0] = client;
    players[1] = client1;
    players[2] = client2;
    players[3] = client3;
}

/////SERVER RUNNABLE/////serverwithgameno
package listserverwithgameno;
/*
 * @author freddiehonohan
 */
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.LinkedList;
import java.util.logging.Level;
import java.util.logging.Logger;
public class Server {
    public static boolean listening = false;
    protected static int port; // Port
    private static int totalConnections = 0; // Total connections counter
    protected static int currentConnections = 0; // Current amount of Clients connected
    protected static LinkedList list = new LinkedList(); // Linked list of Clients
    protected static int totalGames = 0; // Total games counter
    protected static int currentGames = 0; // Current games being played
    protected static GUI g;
    protected static int totAndriods = 0;
    protected static int totIPhones = 0;
    protected static int androidsConnected = 0;
    protected static int iPhonesConnected = 0;

    public static void main(String[] args) throws IOException {
        g = new GUI(); // Create Graphical User Interface
        g.setVisible(true); // Show GUI

        ServerSocket serverSocket = null; // Init Server socket
        while (!listening) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                return;
            }
        }

        try {
            serverSocket = new ServerSocket(port); // Start listening
            g.TA.append("Server listening on port: " + port + ";"); // Print info
            System.out.println("Server listening on port: " + port); // Report socket success
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
catch (IOException e) {
    System.err.println("Could not listen on port: " + port);
    g.TA.append("Could not listen on port: " + port + '\n');
    System.exit(-1);
}

while (listening) { // Infinite loop
    Socket s = serverSocket.accept(); // Accept the connection
    totalConnections++; // Increment counter
    currentConnections++; // Increment counter
    g.TC.setText(Integer.toString(totalConnections));
    g.CC.setText(Integer.toString(currentConnections));
    Client c = new Client(s, totalConnections);
    // Init Client object with its socket and ID number
    list.add(c); // Add to Linked List
    new ServerThread(c).start(); // Fire thread
    System.out.println("Thread fired for Client # " + totalConnections);
    g.TA.append("Thread fired for Client # " + totalConnections + '\n');
}

    g.setVisible(false);
    serverSocket.close();
}

static synchronized void tidyList() {
    int i = 0; // Iterator
    while (i < list.size()) { // While not at end of list
        Client temp = (Client) list.get(i); // Store temp Client to check
        if (temp.socket.isClosed()) { // If Client has disconnected
            list.remove(i); // Remove from list
        } else if (temp.playing) { // Or is playing
            list.remove(i); // Remove from List
        } else { // Only increment counter if first Client in list hasn't been removed
            i++;
            // Check next Client
        }
    }
}

static void incUpdate() {
    totalGames++; // Increment counter
    currentGames++; // Increment counter
    g.TG.setText(Integer.toString(Server.totalGames)); // Update GUI
    g.CG.setText(Integer.toString(Server.currentGames)); // Update GUI
}

static void decUpdate(int i) {
    currentGames--; // Increment counter2
    currentConnections=currentConnections-1;
    g.TG.setText(Integer.toString(Server.totalGames)); // Update GUI
    g.CG.setText(Integer.toString(Server.currentGames)); // Update GUI
    g.CC.setText(Integer.toString(Server.currentConnections)); // Update GUI
}

static void startGame(Game game){
    totalGames++;
    currentGames++; // Increment counter
    g.TG.setText(Integer.toString(Server.totalGames)); // Update GUI
    g.CG.setText(Integer.toString(Server.currentGames)); // Update GUI
    System.out.println("Game # " + game.gameNo + " started");
    g.TA.append("Game # " + game.gameNo + " started" + '\n'); // Print info
}

static void endGame(Game game){
    currentGames--;
    currentConnections=currentConnections-(game.players.length); // Decrement by the amount playing
    g.TG.setText(Integer.toString(Server.totalGames)); // Update GUI
    g.CG.setText(Integer.toString(Server.currentGames)); // Update GUI
    g.CC.setText(Integer.toString(Server.currentConnections)); // Update GUI
    System.out.println("Game # " + game.gameNo + " finished");
    g.TA.append("Game # " + game.gameNo + " finished" + '\n'); // Print info
    switch(game.players.length){
        case 2:
            System.out.println("Client # " + game.players[0].ID + " + " + game.players[1].ID + " disconnected" + '\n'); // Print info
            g.TA.append("Client # " + game.players[0].ID + " + " + game.players[1].ID + " disconnected" + '\n'); // Print info
            break;
        case 3:
            System.out.println("Client # " + game.players[0].ID + " + " + game.players[1].ID + " + " + game.players[2].ID + " disconnected" + '\n'); // Print info
            break;
        default:
            System.out.println("Client # " + game.players[0].ID + " + " + game.players[1].ID + " + " + game.players[2].ID + " disconnected" + '\n'); // Print info
            break;
    }
}
```java
System.out.println("Client # " + game.players[0].ID + " + " + game.players[1].ID + " + " + game.players[2].ID + " disconnected" + '\n'); // Print info
break;
}
```

```java
if (c.clientType==2){ // If Android  
totAndroids++; // Increment counters  
androidsConnected++;  
g.Android.setText(Integer.toString(androidsConnected)); // Update GUI
}
if (c.clientType==3){ // If iPhone  
totIPhones++; // Increment counters  
iPhonesConnected++;  
g.iPhones.setText(Integer.toString(iPhonesConnected)); // Update GUI
}
```

```java
if (c.clientType==2){ // If Android  
totAndroids--; // Decrement counters  
androidsConnected--;  
g.Android.setText(Integer.toString(androidsConnected)); // Update GUI
}
if (c.clientType==3){ // If iPhone  
totIPhones--; // Decrement counters  
iPhonesConnected--;  
g.iPhones.setText(Integer.toString(iPhonesConnected)); // Update GUI
}
```
Server.updateClientStats(client, '*');

switch (client.clientType) {
    case 1:
        System.out.println("Client # " + client.ID + " is a Computer");
        Server.g.TA.append("Client # " + client.ID + " is a Computer" + '\n');
        break;
    case 2:
        System.out.println("Client # " + client.ID + " is an Android Phone");
        Server.g.TA.append("Client # " + client.ID + " is an Android Phone" + '\n');
        break;
    case 3:
        System.out.println("Client # " + client.ID + " is an iPhone");
        Server.g.TA.append("Client # " + client.ID + " is an iPhone" + '\n');
        break;
}

out.println("Enter Game type:" + 1=2 Player, 2=4 Player
client.gameType = Integer.parseInt(in.readLine()); //Store game type

switch (client.gameType) {
    case 1://Two Player Game
        getPartner(); // Check if a partner is available
        if (client.playing) { // If playing i.e. Found an eligible partner
            Game game = new Game(client, client.partner, Server.totalGames + 1, 1);
            Server.startGame(game); // Update Server
            TwoPlayerGame g = new TwoPlayerGame(game); // Init TwoPlayerGame
            g.run(); // Run TwoPlayerGame
            out.close(); // Close output stream
            in.close(); // Close input stream
            } // Else release Client into memory
            break;
    case 2://Three Player Game
        get2Partners();
        if (client.playing) { // If playing i.e. Found an eligible partner
            Game game = new Game(client, client.partners[0], client.partners[1],
                Server.totalGames + 1, 2);
            Server.startGame(game); // Update Server
            ThreePlayerGame g = new ThreePlayerGame(game);
            g.run(); // Run Game
            out.close(); // Close output stream
            in.close(); // Close input stream
        }
        break;
    case 3://Three Player Game
        get3Partners();
        if (client.playing) { // If playing i.e. Found an eligible partner
            Game game = new Game(client, client.partners[0], client.partners[1],
                client.partners[2], Server.totalGames + 1, 3);
            Server.startGame(game); // Update Server
            FourPlayerGame g = new FourPlayerGame(game);
            g.run(); // Run Game
            out.close(); // Close output stream
            in.close(); // Close input stream
        }
        break;
    case 4:// Four Player Game
        getPartner(); // Check if a partner is available
        if (client.playing) { // If playing i.e. Found an eligible partner
            Game game = new Game(client, client.partner, Server.totalGames + 1, 1);
            Server.startGame(game); // Update Server
            TwoPlayerContinuousGame cG2 = new TwoPlayerContinuousGame(game); // Init TwoPlayerGame
            cG2.run(); // Run TwoPlayerGame
            out.close(); // Close output stream
            in.close(); // Close input stream
        } // Else release Client into memory
        break;
    case 5:// Five Player Game
        get2Partners(); // Check if a partner is available
        if (client.playing) { // If playing i.e. Found an eligible partner
            Game game = new Game(client, client.partners[0], client.partners[1],
                client.partners[2], client.partners[3], Server.totalGames + 1, 4);
            Server.startGame(game); // Update Server
            FourPlayerContinuousGame cG3 = new FourPlayerContinuousGame(game); // Init FourPlayerGame
            cG3.run(); // Run FourPlayerGame
            out.close(); // Close output stream
            in.close(); // Close input stream
        } // Else release Client into memory
        break;
}
Found an eligible partner
    Game game = new Game(client, client.partners[0], client.partners[1],
    Server.totalGames + 1, 2);
    Server.startGame(game); // Update Server
cG3 = new ThreePlayerContinuousGame(game); // Init TwoPlayerGame
cG3.run(); // Run TwoPlayerGame
    out.close(); // Close output stream
    in.close(); // Close input stream
} // Else release Client into memory
    break;

  case 6:// Check if a partner is available
    if (client.playing) { // If playing i.e. Found an eligible partner
      Game game = new Game(client, client.partners[0], client.partners[1],
      client.partners[2], Server.totalGames + 1, 3);
      Server.startGame(game); // Update Server
cG4 = new FourPlayerContinuousGame(game); // Init TwoPlayerGame
cG4.run(); // Run TwoPlayerGame
      out.close(); // Close output stream
      in.close(); // Close input stream
    } // Else release Client into memory
      break;

  case 7://
    String s1 = in.readLine();
    String s2[] = new String[2];
    s2 = s1.split("/");
    System.out.println(s2[1]);
    client.stringIP = s2[1];
    client.ipAdd = InetAddress.getByName(s2[1]); // Store Client type
    getPartner(); // Check if a partner is available
    if (client.playing) { // If playing i.e. Found an eligible partner
      Game game = new Game(client, client.partners[0], Server.totalGames + 1, 1);
      Server.startGame(game); // Update Server
cgUDP = new udpGame(game); // Init TwoPlayerGame
cgUDP.run(); // Run TwoPlayerGame
      out.close(); // Close output stream
      in.close(); // Close input stream
    } // Else release Client into memory
      break;

  default:
    Server.totalGames++;// Increment counter
    Server.currentGames++;// Increment counter
    Server.g.TG.setText(Integer.toString(Server.totalGames));
    Server.g.CG.setText(Integer.toString(Server.currentGames));
    client.gameNo = Server.totalGames; // TwoPlayerGame ID
    System.out.println("Game # " + Server.totalGames + " started");
    System.out.println("Games running = " + Server.currentGames);
    Server.g.TA.append("Game # " + Server.totalGames + " started" + \\
    "
    ");
    Server.g.TA.append("Games running = " + Server.currentGames + \\
    "
    ");
    Server.currentConnections--;
    Server.currentGames--;
    System.out.println("Game # " + client.gameNo + " finished");
    System.out.println("Clients connected = " + Server.currentConnections);
    System.out.println("Games running = " + Server.currentGames);
    Server.g.TA.append("Game # " + client.gameNo + " finished" + \\
    "
    ");
    Server.g.TA.append("Clients connected = " + Server.currentConnections + \\
    "
    ");
    Server.g.TA.append("Games running = " + Server.currentGames + \\
    "
    ");
    out.close(); // Close output stream
    in.close(); // Close input stream
    break;
  }
} catch (IOException e) {
    e.printStackTrace();
}
private synchronized void getPartner() {
    Server.tidyList(); // Clean up List of disconnections and Clients playing
    for (int i = 0; i < Server.list.size(); i++) { // Iterate through list
        Client temp = (Client) Server.list.get(i); // Store Temp Client
        if (!temp.equals(client) // If Temp isn't this Client
                && !temp.equals(null) //And it is valid
                && !temp.playing && //And not playing a TwoPlayerGame
                temp.gameType == client.gameType) { //And same gametype
            temp.setPartner(client); // Set this Client as its partner
            client.setPartner(temp); // Set it as this Clients partner
        }
    }
}

private synchronized void get2Partners() {
    Client[] clientArray1 = new Client[2];
    Server.tidyList(); // Clean up List of disconnections and Clients playing
    int counter = 0; // Client needs three partners
    for (int i = 0; i < Server.list.size(); i++) { // Iterate through list
        Client temp = (Client) Server.list.get(i); // Store Temp Client
        if (!temp.equals(client) // If Temp isn't this Client
                && !temp.equals(null) //And it is valid
                && !temp.playing && //And not playing a Game
                temp.gameType == client.gameType //And same gametype
                && counter != 2) { // And all partners have not been found
            clientArray1[counter] = temp;
            counter++;
        }
    }
    if (counter == 2) {
        Client p1 = client;
        Client p2 = clientArray1[0];
        Client p3 = clientArray1[1];
        p1.setPartners(clientArray1, 2); // Set array as its partners
        Client[] ClientArray2 = {p1, p3};
        p2.setPartners(ClientArray2, 2);
        Client[] ClientArray3 = {p1, p2};
        p3.setPartners(ClientArray3, 2);
    }
}

private synchronized void get3Partners() {
    Client[] clientArray1 = new Client[3];
    Server.tidyList(); // Clean up List of disconnections and Clients playing
    int counter = 0; // Client needs three partners
    for (int i = 0; i < Server.list.size(); i++) { // Iterate through list
        Client temp = (Client) Server.list.get(i); // Store Temp Client
        if (!temp.equals(client) // If Temp isn't this Client
                && !temp.equals(null) //And it is valid
                && !temp.playing && //And not playing a Game
                temp.gameType == client.gameType //And same gametype
                && counter != 3) { // And all partners have not been found
            clientArray1[counter] = temp;
            counter++;
        }
    }
    if (counter == 3) {
        Client p1 = client;
        Client p2 = clientArray1[0];
        Client p3 = clientArray1[1];
        Client p4 = clientArray1[2];
        p1.setPartners(clientArray1, 3); // Set array as its partners
        Client[] ClientArray2 = {p1, p3, p4};
        p2.setPartners(ClientArray2, 3);
        Client[] ClientArray3 = {p1, p2, p4};
        p3.setPartners(ClientArray3, 3);
        Client[] ClientArray4 = {p1, p2, p3};
        p4.setPartners(ClientArray4, 3);
    }
}

/////////////////////////////////////////////////////////////////////////
// TWO PLAYER TURN BASED GAME THREAD/-----------------------------------
package listserverwithgameno;

/**
 * @author freddiehonorhan
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TwoPlayerGame {
    private Client player1;
    private Client player2;
    private Game game;
    private boolean running = true;
    private String moveOver = "EOM";

    TwoPlayerGame(Game g) {
        game = g;
        player1 = g.players[0];
        player2 = g.players[1];
    }

    public void run() {
        try {
            // Open Server to Player1 stream
            PrintWriter servToP1 = new PrintWriter(player1.socket.getOutputStream(), true);
            // Open Server to Player2 stream
            PrintWriter servToP2 = new PrintWriter(player2.socket.getOutputStream(), true);
            // Open Player1 to Server stream
            BufferedReader p1ToServ = new BufferedReader(new InputStreamReader(player1.socket.getInputStream()));
            // Open Player2 to Server stream
            BufferedReader p2ToServ = new BufferedReader(new InputStreamReader(player2.socket.getInputStream()));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                Logger.getLogger(TwoPlayerGame.class.getName()).log(Level.SEVERE, null, ex);
            }
            // Notify Client it is first, and its game number
            servToP1.println(player1.playerNumber + " : " + player1.gameNo);
            // Notify Client it is second, and its game number
            servToP2.println(player2.playerNumber + " : " + player2.gameNo);
            String input;
            while (running) {
                input = p1ToServ.readLine();// Read Player1's move
                servToP2.println(input);// Send move to Player2
                /*
                 * while(!input.endsWith(moveOver)){
                 * input = p1ToServ.readLine();// Read Player1's move
                 * servToP2.println(input);// Send move to Player2
                 * if (input.equals("X")) /* If the TwoPlayerGame is over
                 * running = false;
                 * servToP1.println("X"); // Notify Player1
                 * System.err.println("Sent X");
                 * break;// Stop service for both Players
                 * }
                 *
                 * if (input.equals("X")) /* If the TwoPlayerGame is over
                 * running = false;
                 * servToP1.println("X"); // Notify Player1
                 * System.err.println("Sent X");
                 * break;// Stop service for both Players
                 *
                 */
                if (input.equals("X")) /* If the TwoPlayerGame is over
                    running = false;
                    servToP1.println("X"); // Notify Player1
                    System.err.println("Sent X");
                    break;// Stop service for both Players
                }
            }
        }
    }
}
input = p2ToServ.readLine(); // Read Player2's move
servToP1.println(input); // Send move to Player1

/*
while(!input.endsWith(moveOver)){
input = p2ToServ.readLine(); // Read Player1's move
servToP1.println(input); // Send move to Player2
if (input.equals("X")) { // If the TwoPlayerGame is over
    running = false;
servToP1.println("X"); // Notify Player1
    System.err.println("Sent X");
    break; // Stop service for both Players
}
*/

if (input.equals("X")) { // If the TwoPlayerGame is over
    running = false;
servToP2.println("X"); // Notify Player2
    System.err.println("Sent X");
    break; // Stop service for both Players
}

// Notify Server of the disconnects
Server.updateClientStats(player1, '-');
Server.updateClientStats(player2, '-');

Server.endGame(game); // Update Server
servToP1.close(); // Close all streams
servToP2.close();
plToServ.close();
p2ToServ.close();
closeSockets(); // Close both sockets
} catch (IOException ex) {
    Logger.getLogger(TwoPlayerGame.class.getName()).log(Level.SEVERE, null, ex);
}

private void closeSockets() {
    try {
        player1.socket.close();
        player2.socket.close();
    } catch (IOException ex) {
        Logger.getLogger(TwoPlayerGame.class.getName()).log(Level.SEVERE, null, ex);
    }
}

package listserverwithgameno;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
* @author freddiehonohan
*/
public class FourPlayerContinuousGame {
    private Client player1;
    private Client player2;
    private Client player3;
    private Client player4;
    private Game game;
    public boolean running = true;

    FourPlayerContinuousGame(Game g) {
        game = g;
        player1 = g.players[0];
        player2 = g.players[1];
        player3 = g.players[2];
        player4 = g.players[3];
    }
}
player4 = g.players[3];

public void run() {
    try {
        //PlayerToPlayer p1ToP2 = new PlayerToPlayer(player1, player2);
        //PlayerToPlayer p2ToP1 = new PlayerToPlayer(player1, player2);
        PrintWriter servToP1 = new PrintWriter(player1.socket.getOutputStream(), true);
        // Open Server to Player2 stream
        PrintWriter servToP2 = new PrintWriter(player2.socket.getOutputStream(), true);
        // Open Server to Player3 stream
        PrintWriter servToP3 = new PrintWriter(player3.socket.getOutputStream(), true);
        // Open Server to Player3 stream
        PrintWriter servToP4 = new PrintWriter(player4.socket.getOutputStream(), true);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            Logger.getLogger(TwoPlayerGame.class.getName()).log(Level.SEVERE, null, ex);
        }
        // Notify Client it is first, and its game number
        servToP1.println(player1.playerNumber + ":" + player1.gameNo);
        // Notify Client it is second, and its game number
        servToP2.println(player2.playerNumber + ":" + player2.gameNo);
        // Notify Client it is third, and its game number
        servToP3.println(player3.playerNumber + ":" + player3.gameNo);
        // Notify Client it is third, and its game number
        servToP4.println(player4.playerNumber + ":" + player4.gameNo);
        PlayerToPlayers p1ToPlayers = new PlayerToPlayers(player1, player1.partners, game);
        PlayerToPlayers p2ToPlayers = new PlayerToPlayers(player2, player2.partners, game);
        PlayerToPlayers p3ToPlayers = new PlayerToPlayers(player3, player3.partners, game);
        PlayerToPlayers p4ToPlayers = new PlayerToPlayers(player4, player4.partners, game);
        p1ToPlayers.start();
        p2ToPlayers.start();
        p3ToPlayers.start();
        p4ToPlayers.start();
        try {
            p1ToPlayers.join();
            p2ToPlayers.join();
            p3ToPlayers.join();
            p4ToPlayers.join();
            // When threads die game is over...
            // Notify Server of the disconnects
            Server.updateClientStats(player1, '-');
            Server.updateClientStats(player2, '-');
            Server.updateClientStats(player3, '-');
            Server.updateClientStats(player4, '-');
            Server.endGame(game); // Update Server
            servToP1.close(); // Close all streams
            servToP2.close();
            servToP3.close();
            servToP4.close();
        } catch (InterruptedException e) {
            Logger.getLogger(TwoPlayerContinuousGame.class.getName()).log(Level.SEVERE, null, e);
            System.err.println("Interrupted!");
        }
        catch (IOException ex) {
            Logger.getLogger(TwoPlayerContinuousGame.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

/******************PLAYER TO PLAYERS COMMUNICATION THREAD*******************/
package listservewithgameno;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.logging.Level;

package listservewithgameno;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author freddiehonohan
 */
class PlayerToPlayers extends Thread {
    private Client sender;
    private int noOfPartners;
    private Client receiver1;
    private Client receiver2;
    private Client receiver3;
    private Game game;
    public boolean running = true;
    protected PrintWriter servToP2;
    protected PrintWriter servToP3;
    protected PrintWriter servToP4;
    PlayerToPlayers(Client player1, Client[] partners, Game game) {
        sender = player1;
        noOfPartners = partners.length;
        if (noOfPartners == 2) {
            receiver1 = partners[0];
            receiver2 = partners[1];
        } else if (noOfPartners == 3) {
            receiver1 = partners[0];
            receiver2 = partners[1];
            receiver3 = partners[2];
        }
    }

    @Override
    public void run() {
        try {
            // Open Player1 to Server stream
            BufferedReader p1ToServ = new BufferedReader(new InputStreamReader(sender.socket.getInputStream()));
            // Open Server to Player2 stream
            if (noOfPartners == 2) {
                servToP2 = new PrintWriter(receiver1.socket.getOutputStream(), true);
                servToP3 = new PrintWriter(receiver2.socket.getOutputStream(), true);
            } else if (noOfPartners == 3) {
                servToP2 = new PrintWriter(receiver1.socket.getOutputStream(), true);
                servToP3 = new PrintWriter(receiver2.socket.getOutputStream(), true);
                servToP4 = new PrintWriter(receiver3.socket.getOutputStream(), true);
            }
            // PrintWriter servToP1 = new PrintWriter(sender.socket.getOutputStream(), true);
            // PrintWriter servToP1a = new PrintWriter(sender.socket.getOutputStream(), true);
            String input;
            while (running) {
                input = p1ToServ.readLine();
                if (noOfPartners == 2) {
                    servToP2.println(input);
                    servToP3.println(input);
                } else if (noOfPartners == 3) {
                    servToP2.println(input);
                    servToP3.println(input);
                    servToP4.println(input);
                }
                if (input.equals("X") || input.equals("XX")) {
                    running = false;
                }
            }
        } catch (IOException ex) {
            Logger.getLogger(PlayerToPlayer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

//////////// UDP GAME THREAD /////////////
package listserverwithgameno;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class udpGame {
    private Client player1;
    private Client player2;
    private Game game;
    public boolean running = true;

    udpGame(Game g) {
        game = g;
        player1 = g.players[0];
        player2 = g.players[1];
    }

    public void run() {
        try {
            PrintWriter servToP1 = new PrintWriter(player1.socket.getOutputStream(), true);
            PrintWriter servToP2 = new PrintWriter(player2.socket.getOutputStream(), true);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                Logger.getLogger(TwoPlayerGame.class.getName()).log(Level.SEVERE, null, ex);
            }
            // Notify Client it is first, and its game number
            servToP1.println(player1.playerNumber + "=" + player1.gameNo);
            // Notify Client it is second, and its game number
            servToP2.println(player2.playerNumber + "=" + player2.gameNo);
            servToP1.println(player2.stringIP);
            servToP2.println(player1.stringIP);

            String decodedData;
            String[] split = new String[3];
            int newPacketNo;

            DatagramSocket udpSocket = new DatagramSocket(5555);
            byte[] data = new byte[128];
            DatagramPacket sentPacket = new DatagramPacket(data, data.length);
            DatagramPacket forwardPacket;

            boolean player1Finished = false;
            boolean player2Finished = false;

            while (running) {
                udpSocket.receive(sentPacket);
                if (!player1.stringIP.equals(sentPacket.getAddress().toString())&&!player1Finished){
                    decodedData = new String(sentPacket.getData());
                    split = decodedData.split("=");
                    newPacketNo = Integer.parseInt(split[0]);
                    if ((newPacketNo < player1.lastPacket)&&(newPacketNo > 900)) {
                        data = sentPacket.getData();
                        forwardPacket = new DatagramPacket(data, data.length, player2.ipAdd, 4444);
                        udpSocket.send(forwardPacket);
                        player1.lastPacket = newPacketNo;
                    }
                }
            }
        }
    }
}
if(split[1].equals("X")||split[1].equals("XX")||split[1].equals(null)){
    player1Finished=true;
    System.err.println("Player number 1 finished sending");
}
else{
    decodedData=new String(sentPacket.getData());
    split=decodedData.split(":");
    newPacketNo=Integer.parseInt(split[0]);
    if((newPacketNo>player2.lastPacket)||(player2.lastPacket>900&&newPacketNo<100)){
        data=sentPacket.getData();
        forwardPacket = new DatagramPacket(data, data.length, player1.ipAdd, 4444);
        udpSocket.send(forwardPacket);
        player2.lastPacket=newPacketNo;
    }
    if(split[1].equals("X")||split[1].equals("XX")||split[1].equals(null)){
        player2Finished=true;
        System.err.println("Player number 2 finished sending");
    }
    if(player1Finished&&player2Finished){
        running=false;
    }
}
if(player1.socket.isClosed()||player2.socket.isClosed()){running=false;
}
udpSocket.close();
// Notify Server of the disconnects
Server.updateClientStats(player1, '-');
Server.updateClientStats(player2, '-');
Server.endGame(game); // Update Server
servToP1.close(); // Close all streams
servToP2.close();
} catch (IOException ex) {
    Logger.getLogger(udpGame.class.getName()).log(Level.SEVERE, null, ex);
}
private void closeSockets() {
    try {
        player1.socket.close();
        player2.socket.close();
    } catch (IOException ex) {
        Logger.getLogger(udpGame.class.getName()).log(Level.SEVERE, null, ex);
    }
}
ANDROİD FILES

/////////// MANIFEST FILE /////////////
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.freddie"
   android:versionCode="1"
   android:versionName="1.0">
  <application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name="main"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>

      <activity android:name=".SNL"
        android:keepScreenOn="true"
        android:screenOrientation="portrait"/>
      <activity android:name=".SNLGame"
        android:keepScreenOn="true"
        android:screenOrientation="portrait"/>
      <activity android:name=".SNLGame3"
        android:keepScreenOn="true"
        android:screenOrientation="portrait"/>
      <activity android:name=".SNLGame4"
        android:keepScreenOn="true"
        android:screenOrientation="portrait"/>
      <activity android:name=".demo"
        android:keepScreenOn="true"
        android:screenOrientation="landscape"/>
      <activity android:name=".Ball2"
        android:keepScreenOn="true"
        android:screenOrientation="landscape"/>
      <activity android:name=".Ball3"
        android:keepScreenOn="true"
        android:screenOrientation="landscape"/>
      <activity android:name=".Ball4"
        android:keepScreenOn="true"
        android:screenOrientation="landscape"/>
      <activity android:name=".UDP_Ball"
        android:keepScreenOn="true"
        android:screenOrientation="landscape"/>
      <activity android:name=".disp2"
        android:keepScreenOn="true"
        android:screenOrientation="landscape"/>
    </activity>
  </application>

  <uses-permission android:name="android.permission.INTERNET" />
</manifest>

/////////// DEMO ACTIVITY LAYOUT XML /////////////
<LinearLayout android:id="@+id/widget39" android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical"
   android:background="#ff00cccc"
   xmlns:android="http://schemas.android.com/apk/res/android">
  <Button android:id="@+id/P2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="2 - Player"
        android:textSize="30sp"
        android:textStyle="bold"
        android:textColor="#ff009999"
        android:layout_weight="1"
        android:layout_gravity="center_horizontal"/>
</LinearLayout>
<Button android:id="@+id/P3" android:layout_width="fill_parent" android:layout_height="wrap_content" android:text="3 - Player" android:textSize="30sp" android:textStyle="bold" android:textColor="#ff009999" android:layout_weight="1" android:layout_gravity="center_horizontal" />
</LinearLayout>
package com.freddie;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Random;
import android.app.Activity;
import android.app.ProgressDialog;
import android.content.Context;
import android.content.Intent;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.OvalShape;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import android.view.Display;
import android.view.KeyEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.view.Window;
import android.view.View.OnClickListener;
import android.view.MotionEvent;
import android.widget.Toast;

public class SNLGame extends Activity {
    public ProgressDialog dialog;
    public Disc d;
    public Panel p;
    private Handler handler = new Handler();
    private int dispX;
    private int dispY;
    private String serverIpAddress = "10.0.2.2";
    private boolean connected = false;
boolean sent = false;
public int roll;
public String rollOpponent;
public Random die = new Random();
public boolean yourTurn = false;
public int playerNo;

private OnTouchListener TouchListener = new OnTouchListener() {
    @Override
    public boolean onTouch(View arg0, MotionEvent arg1) {
        // TODO Auto-generated method stub
        if (d1.position != 36 && yourTurn) {
            handler.post(new Runnable() {
                @Override
                public void run() {
                    int r;
                    r = die.nextInt(6) + 1;
                    Context context = getApplicationContext();
                    roll = r;
                    CharSequence text = "You rolled a " + roll;
                    int duration = Toast.LENGTH_SHORT;
                    MediaPlayer player;
                    player = MediaPlayer.create(context, R.raw.dice_1);
                    player.setLooping(false); // Set looping
                    Toast toast = Toast.makeText(context, text, duration);
                    toast.show();
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    player.stop();
                    player.release();
                    sent = true;
                    yourTurn = false;
                }
            });
            // p.invalidate();
            return false;
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        Display display = getWindowManager().getDefaultDisplay();
        dispX = display.getWidth();
        dispY = display.getHeight();
        p = new Panel(this);
        p.setOnTouchListener(TouchListener);
        setContentView(p);
        Thread gThread = new Thread(new gameThread());
        gThread.start();
    }

    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_BACK) {
            out.println("X");
            input = "X";
            gameOver = true;
        }
        return super.onKeyDown(keyCode, event);
    }

    public PrintWriter out;
    public String input;
    public boolean gameOver;
public class gameThread implements Runnable {
    @Override
    public void run() {
        InetAddress serverAddr;
        try {
            serverAddr = InetAddress.getByName(serverIpAddress);
            Log.d("ClientActivity", "C: Connecting...");
            Socket socket = new Socket(serverAddr, 4444);
            connected = true;
            handler.post(new Runnable() {
                @Override
                public void run() {
                    Context context = getApplicationContext();
                    CharSequence text = "Connected!";
                    int duration = Toast.LENGTH_SHORT;
                    Toast toast = Toast.makeText(context, text, duration);
                    toast.show();
                }
            });
            Log.d("ClientActivity", "C: Sending command.");
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(new BufferedWriter(socket.getOutputStream(), true);
            // where you issue the commands
            out.println("2"); // Let Server know the type of Client
            // (1=Android)
            input = in.readLine(); // "Enter Game type:
            while (input.equals(null)) {
                input = in.readLine();
            }
            try {
                Thread.sleep(1500);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
            out.println("1"); // Two player game
            handler.post(new Runnable() {
                @Override
                public void run() {
                    dialog = ProgressDialog.show(SNLGame.this, "", "Finding partner...", true);
                }
            });
            input = in.readLine();
            while (input.equals(null)) {
                input = in.readLine();
                try {
                    Thread.sleep(200);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            dialog.dismiss();
            String[] s = new String[2];
            s = input.split(":");
            playerNo = Integer.parseInt(s[0]);
            int gameNo = Integer.parseInt(s[1]);
            if (playerNo == 1) {
                handler.post(new Runnable() {
                    @Override
                    public void run() {
                        Context context = getApplicationContext();
                        CharSequence text = "You are playing 1st!";
                        int duration = Toast.LENGTH_SHORT;
                        Toast toast = Toast.makeText(context, text,
Duration);
        toast.show();
    }}
    try {
        Thread.sleep(1500);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    yourTurn = true;
    while (!sent) {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    out.println(roll);
    for (int i = 0; i < roll; i++) {
        switch (playerNo) {
        case 1:
            d1.move();
            break;
        case 2:
            d2.move();
            break;
        }
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    switch (playerNo) {
    case 1:
        d1.checkRestingPoint();
        break;
    case 2:
        d2.move();
        break;
    }
    sent = false;
} else if (playerNo == 2) {
    handler.post(new Runnable() {
        @Override
        public void run() {
            Context context = getApplicationContext();
            CharSequence text = "You are playing 2nd!";
            int duration = Toast.LENGTH_SHORT;
            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
        }
    });
    try {
        Thread.sleep(1500);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

input = in.readLine();
while (input.equals(null)) {
    input = in.readLine();
}
while (!input.equals("X") && !input.equals(null)) {
    rollOpponent = input;
    int r = Integer.parseInt(rollOpponent);
if (!gameOver) {
    handler.post(new Runnable() {
        @Override
        public void run() {
            Context context = getApplicationContext();
            CharSequence text = "They rolled a " + rollOpponent + "!"
            int duration = Toast.LENGTH_SHORT;
            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
        }
    });
}
try {
    Thread.sleep(500);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
for (int i = 0; i < r; i++) {
    switch (playerNo) {
    case 1:
        d2.move();
        break;
    case 2:
        d1.move();
        break;
    }
    try {
        Thread.sleep(300);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
switch (playerNo) {
    case 1:
        d2.checkRestingPoint();
        gameOver = d2.checkIfWon();
        break;
    case 2:
        d1.checkRestingPoint();
        gameOver = d1.checkIfWon();
        break;
}
if (!gameOver) {
    yourTurn = true;
    while (!sent) {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    out.println(roll);
    for (int i = 0; i < roll; i++) {
        switch (playerNo) {
        case 1:
            d1.move();
            break;
        case 2:
            d2.move();
            break;
        }
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
switch (playerNo) {
    case 1:
        d1.checkRestingPoint();
        break;
    case 2:
        d2.checkRestingPoint();
        break;
}

sent = false;
input = in.readLine();
while (input.equals(null)) {
    input = in.readLine();
}

if (input.equals("X")) {
    handler.post(new Runnable() {
        @Override
        public void run() {
            Context context = getApplicationContext();
            CharSequence text = "GAME OVER... YOU WON!";
            int duration = Toast.LENGTH_SHORT;
            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
        }
    });
    // worthy interesting, 15/20 slides
    // intro
    // background pt iOS stack,
    //
    // iOS/Android focus
    // Nothing specific
    // Concurrency//synchronised
    // 1 sem java
    // Listen to q
    // inherent time stamping client and server
    // Universal server clock
    // UDP mitigate timing problems doesn't guarantee timing
} else {  
    handler.post(new Runnable() {
        @Override
        public void run() {
            Context context = getApplicationContext();
            CharSequence text = "GAME OVER... YOU LOST!";
            int duration = Toast.LENGTH_SHORT;
            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
        }
    });
}

try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
finish();
} catch (UnknownHostException e) {
} catch (IOException e) {
    handler.post(new Runnable() {
        @Override
        public void run() {
            Context context = getApplicationContext();
            CharSequence text = "Cannot find Server!";
            int duration = Toast.LENGTH_SHORT;
            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
        }
    });
}
```java
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

```java
public class dialogRun implements Runnable {
    public void run() {
        try {
            Thread.sleep(4000); // wait 100 ms
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        dialog.dismiss();
        // d.move(16, 15);
        // d.invalidate();
    }
}
```

```java
public Disc d1, d2;
```

```java
class Panel extends View {
    Context c;
    public Panel(Context context) {
        super(context);
        c = context;
        d1 = new Disc(c);
        d2 = new Disc(c);
    }

    int firstDraw = 0;

    @Override
    public void onDraw(Canvas canvas) {
        Bitmap board = BitmapFactory.decodeResource(getResources(), R.drawable.board2);
        // Bitmap board2 = getResizedBitmap(board, 320, 320);
        canvas.drawColor(0Xff00cccc);
        canvas.drawBitmap(board, 0, 62, null);
        Paint p = new Paint();
        p.setColor(Color.WHITE);
        p.setTextSize(30);
        canvas.drawText("SNAKES AND LADDERS", 7, 40, p);
        Bitmap ball1 = BitmapFactory.decodeResource(getResources(), R.drawable.p1); // First Player
        Bitmap ball2 = BitmapFactory.decodeResource(getResources(), R.drawable.p2); // Second Player
        if (firstDraw < 1) {
            canvas.drawBitmap(ball2, d2.posX, d2.posY, null);
            canvas.drawBitmap(ball1, d1.posX, d1.posY, null);
            firstDraw = 1;
        }
        switch (playerNo) {
        case 1:
```
public class Disc extends View {

    public Disc(Context context) {
        super(context);
        c = context;
        squares = new int[36][36];
    }

    public boolean checkIfWon() {
        boolean t = false;
        switch (playerNo) {
            case 1:
                if (d2.position == 36) {
                    t = true;
                }
                break;
            case 2:
                if (d1.position == 36) {
                    t = true;
                }
                break;
        }
        if (t) {
            return false;
        } else {
            return false;
        }
    }

    public Context c;
    public int posX = dispX / 32; // dispX/10;
    public int posY = 64 + (5 * (dispX / 6)); // +dispX/10;
    public int row = 1;
    public int position = 1;
    public int squares[][];

    public void move() {
        if (this.row == 1 || this.row == 3 || this.row == 5) {
            if (position == 6 || position == 18 || position == 30) {
                posY -= (dispX / 6) - dispX / 192;
                row++;
                position++;
            } else {
                posX += (dispX / 6) - dispX / 192;
                position++;
            }
        } else {
            if (position == 12 || position == 24) {
                posY -= (dispX / 6) - dispX / 192;
                row++;
                position++;
            } else {
                if (position != 36) {
                    posX -= (dispX / 6) - dispX / 192;
                    position++;
                }
            }
        }
    }

    p.postInvalidate();
    if (position == 36) {
        MediaPlayer player;
        player = MediaPlayer.create(c, R.raw.move);
        player.setLooping(false); // Set looping
        player.start();
    }
try {
    Thread.sleep(200);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
player.stop();
player.release();

public void checkRestingPoint() {
    boolean action = false;
    boolean snake = false;
    switch (position) {
    case 3:
        posX += (dispX / 6) - dispX / 192;
        posY -= 2 * (dispX / 6) - dispX / 192;
        position = 16;
        row += 2;
        action = true;
        break;
    case 5:
        posX += (dispX / 6) - dispX / 192;
        posY -= (dispX / 6) - dispX / 192;
        position = 7;
        row++;  // corrected action = true;
        break;
    case 12:
        posX += (dispX / 6) - dispX / 192;
        posY += (dispX / 6) - dispX / 192;
        position = 2;
        row--;  // corrected snake = true;
        action = true;
        break;
    case 14:
        posY += (dispX / 6) - dispX / 192;
        position = 11;
        row--;  // corrected snake = true;
        action = true;
        break;
    case 15:
        posX -= 2 * (dispX / 6) - dispX / 192;
        posY -= 2 * (dispX / 6) - dispX / 192;
        position = 25;
        row += 2;
        action = true;
        break;
    case 17:
        posX -= (dispX / 6) - dispX / 192;
        posY += 2 * (dispX / 6) - dispX / 192;
        position = 4;
        row -= 2;
        snake = true;
        action = true;
        break;
    case 18:
        posX -= (dispX / 6) - dispX / 192;
        posY -= (dispX / 6) - dispX / 192;
        position = 20;
        row++;  // corrected action = true;
        break;
    case 21:
        posX += (dispX / 6) - dispX / 192;
        posY -= 2 * (dispX / 6) - dispX / 192;
        position = 32;
        row += 2;
        action = true;
        break;
    case 31:
        //...
posY += 2 * (dispX / 6) - dispX / 192;
position = 19;
row -= 2;
snake = true;
action = true;
break;

case 35:
    posX += (dispX / 6) - dispX / 192;
posY += 2 * (dispX / 6) - dispX / 192;
    position = 22;
    row -= 2;
    snake = true;
    action = true;
    break;
}

p.postInvalidate();
if (action) {
    MediaPlayer player;
    int playback;
    if (snake) {
        player = MediaPlayer.create(c, R.raw.snake);
        playback = 4000;
    } else {
        player = MediaPlayer.create(c, R.raw.ladder);
        playback = 2000;
    }
    player.setLooping(false); // Set looping
    player.start();
    try {
        Thread.sleep(playback);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    player.stop();
    player.release();
}

package com.freddie;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import android.app.Activity;
import android.app.ProgressDialog;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import android.view..
import android.widget.TextView;
import android.widget.Toast;
import android.graphics..
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.content..
public class Ball4 extends Activity implements SensorEventListener {
    
    OrientationEventListener myOrientationEventListener;
    public static final int DIRECTION_RIGHT = 0, DIRECTION_LEFT = 1;
private Panel main;
private Bitmap scratch;
private Canvas c;
public boolean start = true;
private double ballX1 = 10;
private double ballY1 = 10;
private double ballX2 = 10;
private double ballY2 = 10;
private double ballX3 = 10;
private double ballY3 = 10;
private double ballX4 = 10;
private double ballY4 = 10;
private float pitch;
private float roll;
private SensorManager sensorManager = null;
private Handler handler = new Handler();
private String serverIpAddress = "192.168.1.6";
private boolean connected = false;
protected static PrintWriter out;
protected static String data = null;
public ProgressDialog dialog;
private double dispX;
private double dispY;
private int P2dispX;
private int P2dispY;
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    c = new Canvas();
    // ...and the orientation sensor
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
        WindowManager.LayoutParams.FLAG_FULLSCREEN);
    sensorManager.registerListener(this,
        sensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION),
        SensorManager.SENSOR_DELAY_GAME);
    main = new Panel(this);
    Display display = getWindowManager().getDefaultDisplay();
    dispX = display.getWidth();
    dispY = display.getHeight();
    ballX1=(dispX/2)-2*radius;
    ballY1=(dispY/2)-2*radius;
    ballX2=(dispX/2)-2*radius;
    ballY2=(dispY/2)-2*radius;
    ballX3=(dispX/2)-2*radius;
    ballY3=(dispY/2)-2*radius;
    ballX4=(dispY/2)-2*radius;
    ballY4=((dispY/2)-2*radius);
    setContentView(main, new ViewGroup.LayoutParams((int)dispX,(int)dispY));
    Thread gThread = new Thread(new gameThread());
gThread.start();
}

public int playerNo;
public oThread4 outputThread;
public class gameThread implements Runnable {
    @Override
    public void run() {
        Socket socket;
        InetAddress serverAddr;
        try {
            serverAddr = InetAddress.getByName(serverIpAddress);
            Log.d("ClientActivity", "C: Connecting...");
            socket = new Socket(serverAddr, 4444);
            connected = true;
            handler.post(new Runnable() {
                @Override
                public void run() {
                    Context context = getApplicationContext();
                    CharSequence text = "Connected!";
                }
            });
        } catch (IOException e) {
            Log.e("ClientActivity", e.getMessage());
        }
    }
}
int duration = 2000;
Toast toast = Toast.makeText(context, text, duration);
toast.show();

Log.d("ClientActivity", "C: Sending command.");
BufferedReader in = new BufferedReader(
new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(
new BufferedWriter(new OutputStreamWriter(
socket.getOutputStream())), true);

// where you issue the commands
out.println("2"); //Let Server know the type of Client (1=Android)
String input = in.readLine();
while (input.equals(null)) {
    input = in.readLine();
}
try {
    Thread.sleep(1500);
} catch (InterruptedException e) {
    e.printStackTrace();
}
out.println("6"); //Two player game
handler.post(new Runnable() {
    @Override
    public void run() {
        dialog = ProgressDialog.show(Ball4.this, "", 
                "Finding partner...", true);
    }
});
input = in.readLine();
while (input.equals(null)) {
    input = in.readLine();
}
dialog.dismiss();
outputThread = new oThread4();
outputThread.start();
String[] s = new String[2];
s = input.split(";");
playerNo = Integer.parseInt(s[0]);
int gameNo = Integer.parseInt(s[1]);
s = new String[3];
(new Thread(new AnimationLoop())).start();
while (connected) {
    input = in.readLine(); //Read from socket
    if (input.equals("X")) {
        // if gameover
        connected = false; // stop reading
        handler.post(new Runnable() {
            @Override
            public void run() {
                Context context = getApplicationContext();
                CharSequence text = "GAME OVER!";
                int duration = Toast.LENGTH_SHORT;
                Toast toast = Toast.makeText(context, text, duration);
                toast.show();
            }
        });
        out.println("XX"); // Send acknowledgment
        connected = false;
        outputThread.running = false;
        running = false;
    } else if (!input.equals("X")&&!input.equals("XX")) {
        //orientXValue.setText(input);
        //System.out.println(input);// Print if not gameover
        if (input != null) {
            s = input.split(";");
            int playerDataNo = Integer.parseInt(s[0]);
            switch (playerDataNo) {
                case 1:
```java
ballX1 = Double.parseDouble(s[1]) * dispX;
ballY1 = Double.parseDouble(s[2]) * dispY;
break;
case 2:
    ballX2 = Double.parseDouble(s[1]) * dispX;
    ballY2 = Double.parseDouble(s[2]) * dispY;
    break;
case 3:
    ballX3 = Double.parseDouble(s[1]) * dispX;
    ballY3 = Double.parseDouble(s[2]) * dispY;
    break;
case 4:
    ballX4 = Double.parseDouble(s[1]) * dispX;
    ballY4 = Double.parseDouble(s[2]) * dispY;
    break;
} // pitch2 = Float.parseFloat(s[0]);
// roll2 = Float.parseFloat(s[1]);
if (input.equals("XX")) {
    // If acknowledgement
    handler.post(new Runnable() {
        @Override
        public void run() {
            Context context = getApplicationContext();
            CharSequence text = "GAME OVER!";
            int duration = Toast.LENGTH_SHORT;
            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
        }
    });
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
}
```
if (pitch>0) {
    if (ballX1<dispX-(2*radius)) {
        ballX1+=pitch/1.5;
    }
}

if (roll>0) {
    if (ballY1>2*radius) {
        ballY1-=roll/1.5;
    }
}

if (roll<0) {
    if (ballY1<dispY-(2*radius)) {
        ballY1-=roll/1.5;
    }
}

    dataX=ballX1/dispX;
    dataY=ballY1/dispY;
    data=playerNo+":"+dataX+":"+dataY;
    break;

    case 2:
    if (pitch<0) {
        if (ballX2>2*radius) {
            ballX2+=pitch/1.5;
        }
    }
    if (pitch>0) {
        if (ballX2<dispX-(2*radius)) {
            ballX2+=pitch/1.5;
        }
    }
    if (roll>0) {
        if (ballY2>2*radius) {
            ballY2-=roll/1.5;
        }
    }
    if (roll<0) {
        if (ballY2<dispY-(2*radius)) {
            ballY2-=roll/1.5;
        }
    }
    dataX=ballX2/dispX;
    dataY=ballY2/dispY;
    data=playerNo+":"+dataX+"":"+dataY;
    break;

    case 3:
    if (pitch<0) {
        if (ballX3>2*radius) {
            ballX3+=pitch/1.5;
        }
    }
    if (pitch>0) {
        if (ballX3<dispX-(2*radius)) {
            ballX3+=pitch/1.5;
        }
    }
    if (roll>0) {
        if (ballY3>2*radius) {
            ballY3-=roll/1.5;
        }
    }
    if (roll<0) {
        if (ballY3<dispY-(2*radius)) {
            ballY3-=roll/1.5;
        }
    }
    dataX=ballX3/dispX;
    dataY=ballY3/dispY;
    data=playerNo+"":"+dataX+":"+dataY;
    break;

    case 4:
    if (pitch<0) {
        if (ballX4>2*radius) {
            ballX4+=pitch/1.5;
        }
    }
    if (pitch>0) {
        if (ballX4<dispX-(2*radius)) {
            ballX4+=pitch/1.5;
        }
    }
    if (roll>0) {
        if (ballY4>2*radius) {
            ballY4-=roll/1.5;
        }
    }
    if (roll<0) {
        if (ballY4<dispY-(2*radius)) {
            ballY4-=roll/1.5;
        }
    }
    dataX=ballX4/dispX;
    dataY=ballY4/dispY;
    data=playerNo+"":"+dataX+":"+dataY;
    break;
if (pitch > 0) {
  if (ballX4 < dispX - (2 * radius)) {
    ballX4 += pitch / 1.5;
  }
}

if (roll > 0) {
  if (ballY4 > 2 * radius) {
    ballY4 -= roll / 1.2;
  }
}

if (roll < 0) {
  if (ballY4 < dispY - (2 * radius)) {
    ballY4 -= roll / 1.2;
  }
}

dataX = ballX4 / dispX;
dataY = ballY4 / dispY;
data = playerNo + "" + dataX + "" + dataY;
break;
}

private synchronized void doDraw(Canvas canvas, Paint paint) {
  if (start) {
    canvas.save();
    //Bitmap ball = BitmapFactory.decodeResource(getResources(), R.drawable.ball);
    //Bitmap ball2 = BitmapFactory.decodeResource(getResources(), R.drawable.ball);
    //Bitmap board2 = getResizedBitmap(board, 320, 320);
    switch (playerNo) {
      case 1:
        paint.setColor(Color.YELLOW);
        canvas.drawCircle((int) ballX4, (int) ballY4, (int) dispX / 25, paint);
        paint.setColor(Color.RED);
        canvas.drawCircle((int) ballX3, (int) ballY3, (int) dispX / 25, paint);
        paint.setColor(Color.GREEN);
        canvas.drawCircle((int) ballX2, (int) ballY2, (int) dispX / 25, paint);
        paint.setColor(Color.BLUE);
        canvas.drawCircle((int) ballX1, (int) ballY1, (int) dispX / 25, paint);
        break;
      case 2:
        paint.setColor(Color.BLUE);
        canvas.drawCircle((int) ballX1, (int) ballY1, (int) dispX / 25, paint);
        paint.setColor(Color.YELLOW);
        canvas.drawCircle((int) ballX4, (int) ballY4, (int) dispX / 25, paint);
        paint.setColor(Color.RED);
        canvas.drawCircle((int) ballX3, (int) ballY3, (int) dispX / 25, paint);
        paint.setColor(Color.GREEN);
        canvas.drawCircle((int) ballX2, (int) ballY2, (int) dispX / 25, paint);
        break;
      case 3:
        paint.setColor(Color.GREEN);
        canvas.drawCircle((int) ballX2, (int) ballY2, (int) dispX / 25, paint);
        paint.setColor(Color.BLUE);
        canvas.drawCircle((int) ballX1, (int) ballY1, (int) dispX / 25, paint);
        paintsetColor(Color.YELLOW);
        canvas.drawCircle((int) ballX4, (int) ballY4, (int) dispX / 25, paint);
        paint.setColor(Color.RED);
        canvas.drawCircle((int) ballX3, (int) ballY3, (int) dispX / 25, paint);
        break;
      case 4:
        paint.setColor(Color.RED);
        canvas.drawCircle((int) ballX3, (int) ballY3, (int) dispX / 25, paint);
        paint.setColor(Color.GREEN);
        canvas.drawCircle((int) ballX2, (int) ballY2, (int) dispX / 25, paint);
        paint.setColor(Color.BLUE);
        canvas.drawCircle((int) ballX1, (int) ballY1, (int) dispX / 25, paint);
        paint.setColor(Color.YELLOW);
        canvas.drawCircle((int) ballX4, (int) ballY4, (int) dispX / 25, paint);
        break;
    }
  }
  canvas.restore();
private volatile boolean running = true;

@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        if (running)
            running = false;
        else
            running = true;
    }
    else if (keyCode == KeyEvent.KEYCODE_BACK) {
        out.println("X");
        outputThread.running = false;
    }
    return super.onKeyDown(keyCode, event);
}

class Panel extends View {
    Paint paint;
    public Panel(Context context) {
        super(context);
        paint = new Paint(Color.WHITE);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        doDraw(canvas, paint);
    }
}

class AnimationLoop implements Runnable {
    public void run() {
        while (running)
            try {
                Thread.sleep(30);
            }
            catch (InterruptedException ex) {
            }
        updatePhysics();
        main.postInvalidate();
    }
}

    // This method will update the UI on new sensor events
    public void onSensorChanged(SensorEvent sensorEvent) {
        synchronized (this) {

            if (sensorEvent.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
                  // TODO Auto-generated method stub
            }
            else if (sensorEvent.sensor.getType() == Sensor.TYPE_ORIENTATION) {
                pitch = sensorEvent.values[1];
                roll = sensorEvent.values[2];
            }
        }
    }

    @Override
    public void onAccuracyChanged(Sensor arg0, int arg1) {
        // TODO Auto-generated method stub
    }
}