UPPSALA
UNIVERSITET

UPTEC IT 12 020
Examensarbete 30 hp
Februari 2013

Multiplayer Game Server for Turn-Based Mobile Games in Erlang

Anders Andersson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Multiplayer Game Server for Turn-Based Mobile

Anders Andersson

This master's thesis presents the analysis, design and implementation of a game server. The server is specialized for handling traffic for turn-based games, with additional focus on handling traffic over mobile networks. The content of the analysis includes defining requirements, the theory behind the network stack and choices in concurrency model construction. The design part focuses on creating valuable design documents that work as blueprints for the server implementation. The documentation for two prototype implementations are also included, one with basic functionality and one that can be used in the real world to some extent.

The thesis is motivated by a growing need for a simple and easy-to-use game network solution. The independent game development scene is growing rapidly since the introduction of the modern smart phone, which comes with great gaming capabilities. Independent game developers are often small studios that might not have demands for big, complex and expensive systems. This product aims to solve that problem by providing a simpler and more user friendly solution.

The thesis concludes that the construction of a working game server solution is possible to achieve on a small budget. It also concludes that a game server does not necessarily need advanced features to be functional.

Handledare: Josef Nilsen
Ämnesgranskare: Arnold Pears
Examinator: Olle Ericsson
ISSN: 1401-5749, UPTEC IT12 020

Contents

1 Introduction

1.1 Motivation

The independent game development scene has been growing very rapidly since the release of iPhone in 2007 and is expected to double its total revenue from 2010 to 2015 [2]. This evolution is very interesting, when just a few years ago the gaming market was controlled almost solely by big publishers releasing AAA titles (premier and high-quality games) for computers and consoles with multi-million dollar budgets. The new mobile gaming market is currently dominated by a lot of small developers [26], making the market a lot more diverse and making it easier for new developer teams to reach the market without any major funding.

With this in mind, there are a number of game types that fit well into the mobile gaming platform. According to [17] games can be divided into four different categories; *singleplayer games, pseudo multiplayer games, turn-based multiplayer games* and *real-time multiplayer games.* The article rules out *real-time multiplayer games* for mobiles, since they require a lot of bandwidth and a stable connection; this is something that is not yet present for mobile phones. While 3G (and 4G) networks can provide the same bandwidth as fixed broadband network carriers, the problem lies instead in the round trip time (RTT). The RTT over a 3G network is not only considerably higher (with times ranging between 100 ms and 2000 ms) compared to a fixed broadband, but it's also highly unstable as it fluctuates constantly [23]. This is a big problem for real-time multiplayer games, where users need a fast response time; preferably with a RTT under 50 ms. The constant fluctuation is also a problem. Ponder for example a network where the RTT was fixed at 300 ms. Users could then adapt to these circumstances, and plan their moves with this delay in mind. While still not optimal, the game could at least be playable. In absence of RTT guarantees, playing real-time games over a 3G network does not sound very compelling. A possibility to enable a real-time game to be played on a mobile phone is to only let the game be played when the phone is connected to a fixed broadband via its Wi-Fi connector. This does, however, ruin some of the interesting aspects of playing games on a phone; to be able to play whenever you want, wherever you are.

The RTT problem is one of the main reasons for the choice of building a server for handling *turn-based* gaming traffic. A turn-based game does not suffer from delays and packet loss. A user will be satisfied with numbers way worse than what a 3G network provides. The other reason is that turn-based gameplay is very well suited to mobile phones, since the user can check his phone for short periods of time several times a day to make his moves, in contrast to playing a real-time game where the user must dedicate a large portion of time in one sitting. This aspect is important since it makes use of the revolutionary new mobile gaming platform; a large portion of the population are always carrying their gaming device with them without having to think about it.

The two big mobile markets to distribute games on are currently App Store for iOS and Android Market for Android who together hold a marketshare of over 65% [4]. The most popular multi-player games in Sweden are currently *Wordfeud* and *Draw Something*; both of which are turn-based and have been downloaded millions of times

world wide. In conclusion, there is definitely a market for these sorts of games.

1.2 Erlang

Erlang is the language of choice for the implementation of the server system. This is motivated by Erlang being a very powerful language for distributed, fault-tolerant, real-time, concurrent systems [16]. Erlang was originally developed for use in telecommunication systems by Ericsson in the 1980's and the characteristics of the language place considerable emphasis on run-time safety, concurrency and distribution. These characteristics are highly relevant in today's server environment. An important thing to note is that Erlang was, unlike most popular languages, developed with the idea of solving a particular problem. This is why Erlang is outstanding for these types of solutions. Erlang is not like the big languages C and Java, where basically anything can be developed with decent results. Erlang should only be used in systems similar to the original telecommunication problem [16].

To mention one of Erlang's several special abilities, the way Erlang handles processes and concurrency is very interesting. An Erlang process is not handled in a conventional way by the Operating Systems scheduler, it is instead handled internally by the Erlang VM. When Erlang operates on a multi-core architecture, one thread per core is created by the Erlang VM, and the processes are balanced across the different cores. This is one of the key features for Erlang's high performance and ease of use concurrency model [16].

1.3 Related Work

Video Gaming is a huge industry, valued at 65 billion US Dollars in June 2011 [1]. It comes as no surprise that there are lots of different game networking solutions in existence today. A handful of the biggest competitors are described in this section.

1.3.1 Apache MINA (and other Java APIs)

The biggest player on the market for network solutions in independent game development is currently Apache MINA [13]. MINA is a network application framework that provide developers with an API for any kind of application requiring some form of sophisticated way for handling network traffic, while helping the developers with the handling of scaling, traffic throttling and overload shielding issues. A server built in MINA generates high performance for being coded in JAVA, on par with solutions coded in C/C++ [5].

There are lots of other comeptitors that also provides a JAVA API, for example Netty and Grizzly. They have in common that they all make use of the New I/O Java APIs (NIO). The reason for choosing Java as the implementation language is that it's well-known among developers, easy to use, it comes with powerful APIs for other areas and the language can be used to implement all possible functionality; whereas Erlang is more specialised. However, this typically comes with a cost in performance.

1.3.2 SmartFoxServer

SmartFoxServer is a more specific network solution that is marketed directly to multiplayer game development and nothing else. On top of its APIs for several popular platforms, it also features tools such as a visual configuration tool for configuring the server. Its APIs include a lot of high-level classes for handling different rooms, zones and game specific events. In an MMORPG for example, rooms and zones can be used as different scopes where several users are connected to each other in real time, while not being connected in the same direct way to the rest of the world. The basic version of the SmartFoxServer can be run without any server-side coding, leaving it up to the clients to implement the game logics [13]. In conclusion, the SmartFoxServer provides the user with very high-level tools to solve a specific task, be it real-time or turn-based games. This leads to a specialized very high performance engine that is easy to use. The downsides are that it focuses intensively on a narrower field compared to using traditional networking APIs, it's not modifiable since it's closed source and it actually costs a great deal of money to get a license.

Other popular game server platforms include ElectroServer, Nonoba and the networking parts of XNA and SteamWorks for example. Those are similar to SmartFoxServer in the way that they provide very high-level APIs for handling game specific tasks.

1.3.3 Publish-Subscribe Pattern

Publish-subscribe is a popular concept, where a server works as a publisher of data and where clients subscribe to the data. This sort of system is very common to use in any type of mobile apps. It commonly uses push messaging as the way to communicate. To relate this concept to game networking, each game can be set up as a publishing channel. Each client in the game subscribes to the data from that channel. This creates a game room in a similar manner as more game specific network solutions such as SmartFoxServer. Some examples of publish-subscribe systems are Pubnub and Pusher.

1.3.4 Apache Tomcat (and other web servers)

Another popular way of implementing a game server is to use a web server (or servlet container). A web server is a platform that assists an application in publishing content that can be accessed through the Internet [21]. A popular choice is Apache Tomcat; a HTTP web server and servlet container that publishes Java content to the Internet. When using Tomcat for implementing a game server, the developer puts his content in the container and tomcat then publishes the content to be accessed via the Internet. The container then handles life-cycle control, scaling and concurrency. Tomcat is particularly easy to use, since the deployment stage is almost automatic. Other web servers include Apache web server and Nginx, which provide the developer with a more general HTTP web server that can be used with any language of choice.

Web servers are generally very optimized and powerful for handling HTTP traffic, and this makes them a good choice for game servers. One should note, however, that a web server is initially intended to publish content on the Web and not to be used as a game server. While the popular web server engines are very fast and provides the user

with powerful features, there are still restrictions in performance for games. A game might not need all the provided functionality that comes with the web server and with the HTTP protocol, thus generating some overhead. Also, a lot of the popular web servers have relatively low limits for concurrent users.

1.4 Thesis Objectives

In this thesis, a multiplayer game server for turn-based mobile games is analysed, designed and implemented. The server is to be used by the company Bitbindery AB as a general purpose server for upcoming game titles to be released for mobile operating systems such as iOS and Android, but might also be used as a stand alone product marketed directly to other game development companies. The server will be high performance, high concurrency, scalable, fault-tolerant and allow for dynamic updates. All those features are well supported by the Erlang programming language.

As we uncovered in the related works section, the existing solutions show a very stiff competition. So, why not use a pre-existing solution? There are solutions, like SmartFoxServer, that have been in development since 2003 [13]. It seems like an impossible task to compete with such giants...

The reason for making an in-house server is that it can be fitted perfectly for the requirements with very little overhead. When using a pre-existing game server (like SmartFoxServer) there will be lots and lots of functionality that is not required and not used. This could possibly lead to performance degradation, but above all it will be less user friendly since the APIs are so extensive. The code is also closed source (in the example of SmartFoxServer), which means that there can be hidden issues, like security holes, that can't be foreseen by the developers. Closed source also puts restrictions on the developers, as they can't extend the server.

Using any of the popular Java APIs (like Apache MINA) is actually on par with using Erlang in terms of abstraction level. Those APIs basically implements the same functionality that Erlang was made for from scratch, so they would not decrease development time and will definitely not perform better, as Erlang proves to be the highest performing language at this abstraction layer for this particular task [16].

1.5 Thesis Outline

The thesis is structured as follows:

- In section 2, the requirements of the system are described, followed by an analysis of how to meet them. Comparisons are made between different transport layer protocols, application layer protocols, push or pull technology, concurrency model and much else.
- Section 3 presents the first prototype implementation. This prototype implements the minimum required functionality to qualify as a server. Basic tests are performed to identify performance limits.
- Section 4 is the design section and it presents the user API, the Erlang Game Server Protocol and the architecture of the system.

- The second prototype is presented in section 5. This is a more refined prototype that can perform selected functions from the design documentation.
- Future extension are explored in section 6. This includes advancements in security, distribution, fault tolerance and documentation.
- Conclusions about the project's outcome are drawn in section 7. This includes describing how well the goals were met and some discussion about the server's impact on the network gaming market.
- The report ends with a summary in section 8.

2 Analysis

This section deals with the different requirements that the system must meet, both functional and non-functional, which architectural model to use and how choices between transport- and application layer protocols are motivated. It also deals with project-specific decisions in regards to performance, availability and robustness, such as limits of different parameters like maximum number of users in a game and maximum number of active games per client.

In this section, and in the rest of the report, three different modules are discussed; the *statistics module*, the *network module* and the *game mechanics module*. The statistics and game mechanics modules are not included in in the actual analysis, design and implementation; they can be seen as black boxes with a simple description of functionality. Only the network module is handled in this report and might be referred to as only *the server* in some sections. When something is referred to as a *client*, it refers to the network interface that is communicating with the server. All other client functionality is outside the scope of this report. A *user* or *end user* in this text means a user that is using an application on a client; a *gamer* if you will. A *game developer* is someone who wants to use the server in his or her game development.

2.1 Requirements

We commence by establishing the necessary requirements for a correct description of the intended server functionality. Two types of requirements are usually considered, functional requirements and non-functional requirements. Functional requirements describe system functionality [24] and this section is kept non-technical, to allow for easy understanding of what services the server delivers. A non-functional requirement is something that describes the operation of the system, rather than the system's function [24], and in this section, performance expectations in terms of availability, performance, security, safety and robustness are explored.

2.1.1 Functional Requirements

The server handles traffic for turn-based games. This means that the server is logically at the nexus of the system, and traffic from every single client must pass through the server as shown in figure 1. To describe in detail what the server accomplishes and what services it provides, a list of the functional requirements for the server is needed:

- Clients can issue a request for creating a user account. The user account includes a username and a password. If the username is not already in use, the new user account will be created, thus generating a new user.
- Users can log in with their account on any client. The users can also log out if they so desire.
- The server implements a player lobby that provides information about all registered users. A client can search a database of all users, and assign a user with other users as friends.

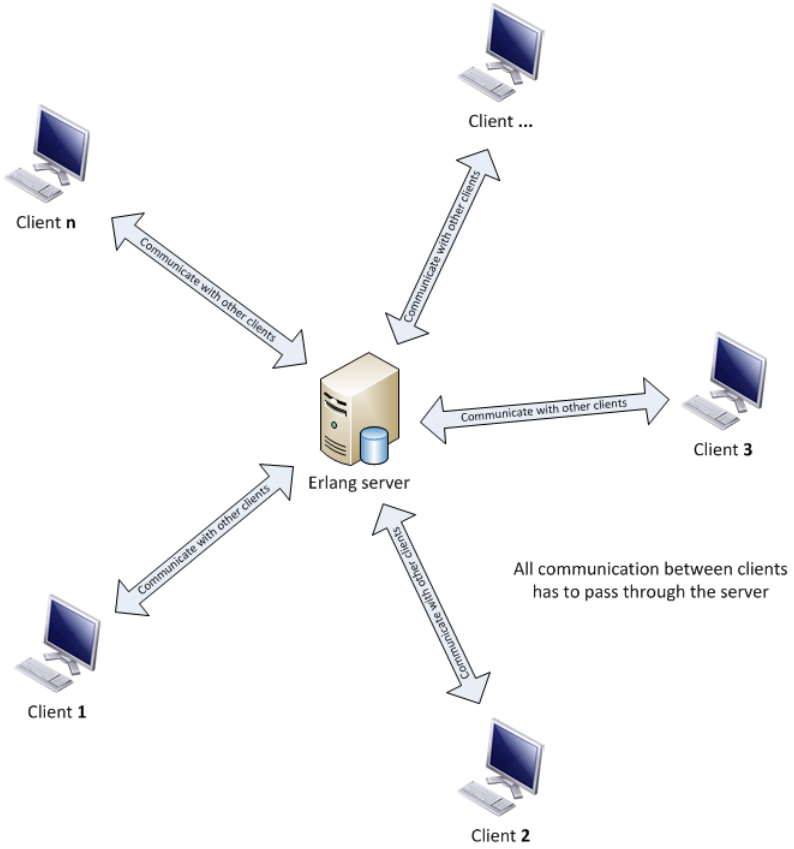

Figure 1: The server handles all traffic that is sent between clients.

- A user can chat with other users. Users can only chat with one user at a time (a two-way chat) and only with users on their friends list. A user can have one chat open per friend.
- Each user in the player lobby system is kept with statistics. The statistics includes rating (ranking) and numbers of won and lost game.
- The network module communicates with a *statistics module* where statistics about user behaviour is handled. This module is easily interchangeable, to allow for game specific statistics.
- The server implements a game lobby that provides information about all active game sessions. Games that have not started yet can contain between 1 and 8 users. Clients can be assigned to those games. Games that have started can contain between 2 and 8 users and can not assign new users.
- Clients can request to start a new game. A game session is then created, with that client as game master. That client can issue commands to set the number of maximum players allowed and a time limit for each move, to start the game, to kick players and to terminate the game lobby.
- When a game session is started, the game lobby is terminated and the game master role is discarded. Then, while playing the game, there is no master; all

players are equal and are free to resign from the game or to make their next move. All other client functionality is handled outside of this report's scope; most possibly in the game mechanics module.

- Clients can request to be put into an *automatic game session assigning system*. The server then assigns players randomly with other players who are using the system. The client can specify if it wants to play in a game with 2 players, 3-4 players or 5-8 players. The client can choose if it wants to play in a ranked or an unranked game.

- When a game session is started, the actual game-play begins. During game-play, the server receives data from a client and sends it to all other clients. For example, if a user was to make a move, his client sends the data necessary to express that move to the server and the server sends a game state update to all clients in the game.

- The server keeps track of which client's turn it is and only accepts game changing data from that particular client.

- The game mechanics are implemented in a separate module that is utterly oblivious to its surroundings. This module is easily interchangeable, to allow the server to run any type of turn-based game. All the game mechanics module does is to take incoming data from the network module, process it and return the processed data to the network module.

- A client can have a maximum of 20 game sessions active at any single point in time.

- The server keeps an updated record of client addresses. Since mobile phones might change IP several times each day (from using different Wi-Fi networks for example) each client must notify the server of its current IP address for the server to be able to communicate with it. The record links together a user's identity with the corresponding address.

Note that a game is not required to make use of all the server functionality. When a developer uses the server for a game, he can choose the functionality that is necessary for that particular game. He might for example not want to include the chat module, the friends-list or the statistics. This is described in more detail in the API part in the design section.

2.1.2 Non-functional Requirements

Performance While the traffic load on the server is relatively low, since each client sends data relatively seldom (compared to a system for handling real-time games for example), effort will still be put into performance for the sake of cost efficiency. Since the server should be able to support tens of thousands of users and games, much effort will be put into performance in terms of network traffic handling, process optimization and memory usage minimization.

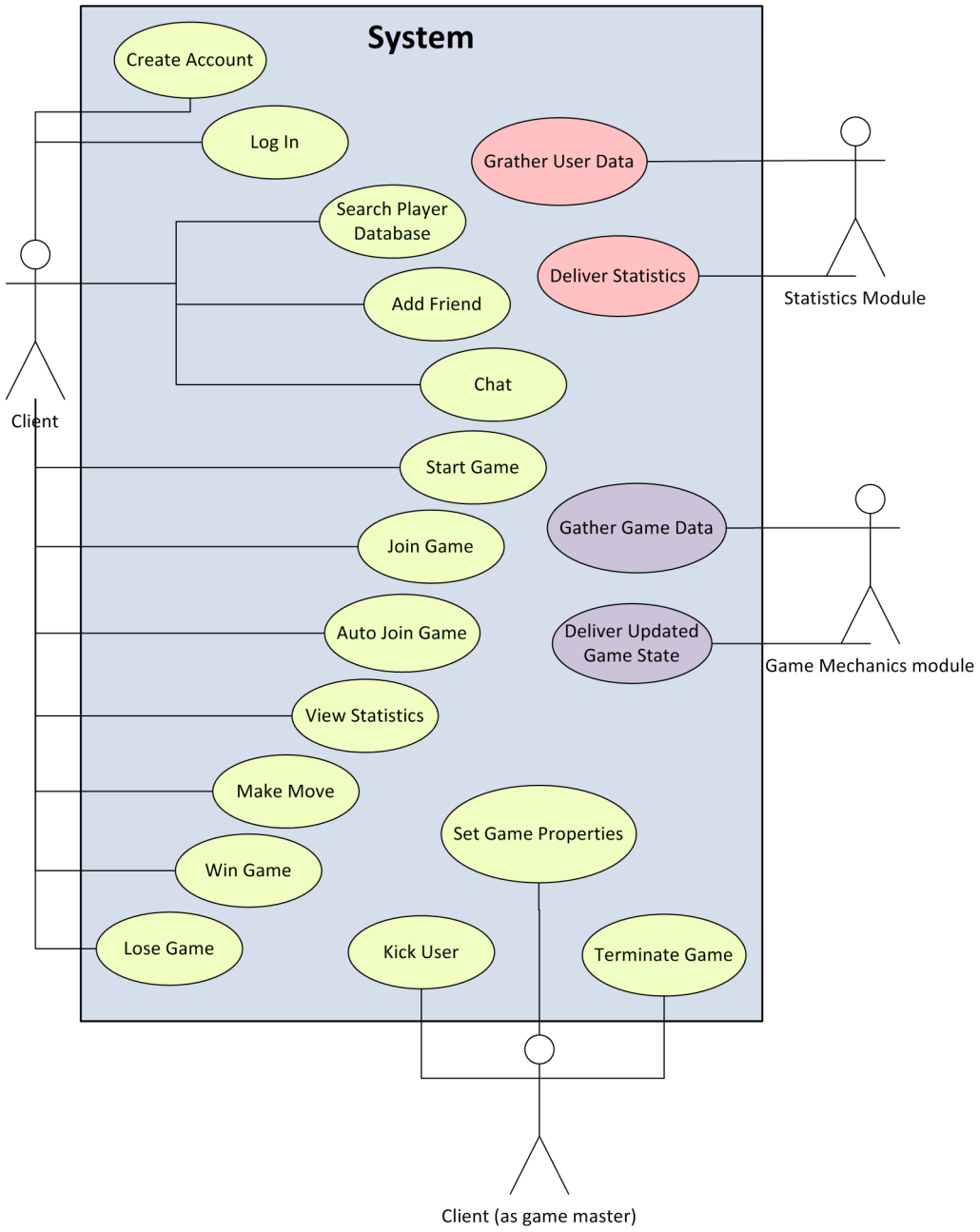

Figure 2: Use case diagram of the system. The *Client (as game master)* role inherits from the *Client* role.

Scalability In some situations the server might not be used for games with many active users and in other situations the server might be used to support tens of thousands of users simultaneously. This makes scalability a very important subject since the server should scale well in order to make full use of its resources. A well designed concurrency model is the key to good scalability.

Usability The system should be easy to use for any developer who is to work on a client application that communicates with the server. The client application program-

mer will be provided with a simple and easy to use Application Programming Interface (API) to interact will all server functionality.

Documentation For this type of technical solution that has a target audience of developers, documentation goes hand in hand with usability. The API and the server components will be well documented for increased usability.

Availability For every system that is handling traffic for hundreds of thousands of users, availability is of great importance. In the web server world the business standard is called *the five nines*, meaning that any uptime less than 99.999% is unacceptable. This is however debated frequently within the community, and it's very hard to give a definitive answer to this. The acceptable uptime rate is commonly stated to be between 98% and 99.999% depending on what business you are in. Since Erlang is fault tolerant and also supports hot code updates (meaning that the server can stay up while performing an update) the server software itself should not pose a big problem to reach very high uptime rates, as long as the code is written in a fault-tolerant way. Other factors such as host downtimes and DDoS-attacks, that also affects availability, are not in the scope of this report.

Robustness To allow for high availability, the server must be robust and be able to handle errors and continue operation when unexpected events occurs.

Extensibility In this report, *The server* is in fact an extensible network module, where other modules will work in co-operation to present the totality of the system. The system must be modular and easy to extend with new module add-ons in the future.

Security A system that is open for any client to connect to and that keeps a database of user information needs to consider some security measures. The code should be written in a safe manner and the traffic and user data needs to be encrypted. Encryption could possibly have an effect on the servers performance, and this will vary depending on what encryption method is used. The security's impact on performance must be weighted carefully.

2.2 TCP or UDP

Choosing what transport layer protocol to use is essential to server functionality. Aside from TCP and UDP, which are the two commonly used protocols in the Internet protocol suite, Erlang is also shipped with an interface for a third transport layer protocol, namely the Stream Control Transmission Protocol (SCTP) [7]. To describe SCTP, it's a good idea to start by analysing the characteristics of TCP and UDP.

UDP is by far the simplest protocol of the three, only providing the minimum of what is needed in the transport layer. Packets of data are simply sent through the IP-environment as datagrams and datagrams are basically only provided with a source and a destination address. No stable connection is initiated via a hand-shake (like TCP), and the datagrams are delivered without any ordering, they might be missing upon arrival or might arrive duplicated [21]. This idea of just dumping packets into

the Internet's stream of data without any control is either motivated by providing a service where each packet is not crucial for application functionality, or where the handling of ordering, resending and time-outs are handled by the application itself in the application layer protocol. Services that don't depend too much on each and every packet are, for example, video-streaming services and real time games. In video streaming, it does not matter too much if some frames in the video are skipped and the pros of having a faster delivery time of packets outweighs the cons of potential loss. The same might apply for parts of real-time games (not turn-based games), where only the latest packet sent is of importance, and where connection control might cause the game to slow down.

TCP is the original transport layer protocol within the Internet protocol suite. It's characterized by providing a connection oriented service and guarantees that each packet is received intact and in the correct order. TCP can make those guarantees by initiating a connection between two parts via a three-way-handshake and by acknowledging that each packet sent during the connection has been received without modification [21]. This comes with a cost, as resending lost packets takes time. This makes TCP slower compared to UDP, but a lot more stable. It's also easier to use TCP in an implementation where each packet is of importance, since the functionality is handled by TCP, and not by the application itself. TCP implements a lot of functionality that is missing in UDP, features such as flow control and congestion control [21]. TCP is used in applications where each byte of data is of importance and where higher latency is of less importance. It's for example used in the World Wide Web, E-mail services and file transfer protocols.

SCTP is a relatively new protocol, first standardized in October 2000 [25]. It is something of a combination of TCP and UDP, combining the message oriented service of UDP with the reliable in-order guarantee and flow control of TCP. While this functionality is comparable to TCP, the interesting differences lies instead in what new services SCTP provides, that are neither found in TCP or UDP; multi-streaming and multi-homing. When using TCP, only one stream of data is handled by one association and each packet in this association is delivered in order. With multi-streaming, each association can instead hold several different streams that are ordered independently from other streams. Multi-homing is the concept of allowing a connection to be handled by multiple network points in either end of the connection. This can be used for increased availability and decreased recovery time from network errors [25].

To make the choice between the three different protocols for the server, the characteristics of the traffic must be established. By looking at the requirement section, a quick overview can be created:

1. The chat traffic will consist of small packages sent between intervals of perhaps 10 seconds to several hours and each packet of data is of great importance, since all text in the message must arrive intact.

2. The traffic generated when searching for active game sessions, users and friends will resemble the traffic produced between a web browser and a web server. The client will request data from the server, the server will perform a data look-up and send the data to the user. All data is of great importance and must be

delivered intact.

3. The turn-based game-play traffic will resemble a chat system from the network's point of view. A client sends a packet of data when making his move (like sending a message), the server receives the data, processes it and sends it out to all clients connected to the game session (like people in a chat room). That the server actually processes the data a great deal to figure out the game state does not impact the traffic's behaviour. All data sent is of great importance and it's sent with large intervals in-between, depending on the game type. A qualified guess would be somewhere between 30 seconds and several hours.

4. The traffic generated by clients notifying the server of their current address might either need a stable connection or not, depending on how its implemented. If the clients send notifications with a short interval, lets say every 5 seconds, then each packet is not of great importance, since it's only the latest packet of data that matters; it's affordable to miss some updates. If the update occurs more seldom however, say every hour, then each packet would be important.

Lets start the transport layer protocol analysis by looking at what features the SCTP protocol might bring to this project. The two big factors are obviously the multi-streaming and the multi-homing services. Multi-streaming is generally used for creating reliable data streams. This is not useful in this application, since all transmissions will consists of short messages sent with large intervals in between; there are no high demand data streams. Multi-homing is also used for stability, since if one network interface fails, the alternative interface can continue operation as if nothing happened. This is also something that is of no use to this application, since mobile phones normally only use one network interface. For the server-side it's very possible that several network interfaces could be used for one connection. This does not however come with any great benefits, since there is no real need to have such an incredibly reliable data stream anywhere in the system. Consequently, SCTP is not a suitable protocol and will not be used in any part of the system.

As already established, TCP is used where reliability and stability is important and UDP is used where performance is critical to operation. In the first three types of traffic (the chat, game lookup and in-game traffic) it's clear that reliability and stability is far more important than performance. It's easy to draw parallels to other services using TCP, such as web servers and chat systems. That being said, UDP is rarely used on its own; it's more common to implement the desired functionality on top of the UDP protocol to match the system requirements. The choices to consider is therefore more about using TCP for the sake of simplicity or to implement a complex solution on top of the UDP protocol. It's obvious that the ordered delivery of data guaranteed by TCP is something that is required for operation and would thus have to be implemented on top of the UDP protocol. This would result in a lot of extra work without any guarantee what so ever that it would actually be more optimized than the mechanisms already available through TCP. The unnecessary work combined with the low demand for high performance leads to the choice of TCP for the first three types of traffic.

For the address update traffic, the choice might depend on implementation. For the

case where updates are sent more seldom, TCP is an easy choice with the same motivation as in the previous paragraph. If updates are sent more often with some degree of acceptable loss, then a pretty naked UDP solution could be viable. However, studies have shown that the packet loss can be incredibly high over 3G-networks, as high as 50% when travelling by train for example [20]. With that much loss, the idea of just spamming updates might not be as good as first thought. It would have worked well in a fixed broadband, but under the mobile conditions it will not. Therefore, TCP will be used also for this type of traffic, to guarantee delivery even when the packet loss is high, without having to implement a guarantee system on top of UDP.

2.3 Application Layer Protocol

The choice of a transport layer protocol can, as explained in the previous section, be motivated in a scientific way. The comparison between TCP and UDP is based mostly on measurable attributes such as performance, stability and reliability. Usability is not taken into consideration because the transport layer protocol is an internal mechanism that will not impact the user experience. While the choice of an application layer protocol is also made partly from hard data such as performance, it's also important to consider usability. The application layer protocol is the part of the protocol stack that is facing out towards the game developers and the choice is therefore greatly affected by previous user experience and user friendliness.

The API (Application Programming Interface) between the transport layer and the application layer is handled by *sockets*. A socket is used as a access point for a process, where data can be sent to and received from some other process in another system. A socket hides the complexity of the transport layer and provides an easy to use interface for the developer [21]. On top of the socket, there is also a need for an application layer protocol. This protocol is like an agreement between the client and the server on how the data should be described. Sockets are described in more details in **section 2.6**.

The most widely used application layer protocol is the HyperText Transfer Protocol (HTTP) [21]. It's used to implement the World Wide Web and is therefore very well known among developers, especially web developers. The intended use of HTTP is for a Web Browser to initiate a connection to a Web Server and to receive data from the server. It's a simple process, where the client sends a *request* to the server using the *GET* method, describing what resources it wants to access; a *web URL* for example. The server then responds by providing the requested resources to the client. The most commonly used data is an HTML-file with some accompanying media-files. This means that the standard use of HTTP includes sending large amounts of data from the server to the client, and this is what HTTP is optimized for doing. Another method, the *POST* method, is used to submit data to the Server from the Client, and may result in the creation of new or updated resources. Note that *GET* and *POST* can be used for other purposes as well. On top of these two methods that are most commonly used, HTTP also defines 7 other methods, such as *DELETE* and *TRACE*. HTTP is a stateless protocol, meaning that the server does not keep track of its connected clients. A state can however be implemented using cookies for example. Some other features of HTTP include the choice between persistent and non-persistent connections, status codes, time-stamps, server and client application versions and Etags.

Using HTTP as the application layer protocol does have some positive parameters for this project. The intended use of HTTP is somewhat related to the game server's architecture where packets of data are sent asynchronously from the server to several client applications. The game developers using the server might also have good knowledge of the protocol. However, HTTP comes with some overhead that is not necessary for the server implementation. As listed above, HTTP comes with a lot of different features that are specific to solving problems for web developers. Another thing to remember is that if HTTP is used, there will still be a need to define a self-invented application layer protocol on top of HTTP, to match the functionality in the game server's API. One positive thing about wrapping data in HTTP-packets is that this, according to various claims, reduces problems with firewalls a great deal, since firewalls are almost always configured to let HTTP traffic through. The negative part is that the information will be wrapped in otherwise unnecessary information that will in turn have a negative impact on the server's performance from increased packet sizes. Consequently, the performance can be increased by using a more light-weight protocol that does only what is needed for the game server's functionality.

For light-weight protocols, where the developer defines the protocols functionality himself, JSON is a popular alternative. JSON is designed to be a light-weight data interchange format that is easy to read and write for humans and easy to generate and parse for computers [11]. Figure 3 shows an example of a packet with the JSON structure. It's similar to XML in the way that the data is human readable in a bracket form. The data in JSON is expressed in nested collections of name/value pairs as shown in the example.

```
{
      "game": "BitNet",
      "action": "Move",
      "parameters"  :
      {
          "X" : "67",
          "Y" : "3",
      },
}
```

Figure 3: Example of a JSON-packet.

Note that JSON is not a protocol, it's a format in which a protocol can be defined. This invokes the question: why use JSON instead of just defining a protocol from scratch? The answer is that JSON is a well defined standard that is used over many different platforms and is supported by many popular programming languages. This means that there are JSON-parsers available for Erlang, Java, C++, Objective-C, C# and many other languages [11]. This covers all popular mobile phone platforms, and

makes it a lot easier to write client API's for them. If a protocol would be constructed from scratch, a parser would have to be implemented for each supported language. Additionally, Erlang is a small language with a market share of only 0.2% [14] and supported protocols can therefore be hard to find. The parser for Erlang is called ejson and it provides full support for the format [8]. In conclusion, JSON is a very fitting format for data-interchange between an Erlang server and clients written in arbitrary languages and will thus be the utilized format for the application layer protocol implementation.

2.4 User API

While the application layer protocol is designed to be human readable and could be used directly by the game developers, an implementation of a user API is still of value, as it makes the communication with the server much more straightforward and lets the game developers work at a higher abstraction layer. The user API will consist of functions that correspond to each client use case established in the requirements section such as creating a new game, joining a game, making a move or sending a chat message. One must note that a user API, contrary to the application layer protocol, must be specifically implemented for each programming language to be used. Thus, the choice of supported languages must be made with care, in respect to what platforms should be supported. The programming languages that do not get an API can still use the server by direct use of the application layer protocol or by implementing their own API. There might be a need for developers to use the application layer protocol directly even in programming languages where an API does exist, as they might want to extend on the server functionality. See the design section for further description and full implementation of the user API.

2.5 Sockets

The basic building block that handles the traffic to and from the server are the sockets. A socket is the same thing as a file that can be read from and written to between remote machines (or within the same machine for that matter) [21] and is something that is essential to use for network communication. The server will use TCP sockets, as the server uses TCP as it's transport layer protocol of choice. The TCP socket comes with an API from which a developer can access the different functions and settings for the protocol. Erlang is shipped with a module called gen_tcp that implements this API [9].

The standard way of using a TCP socket in a client-server architecture is for the server to have the socket listen to connection attempts from clients. The server starts a socket in listening mode and can then start accepting incoming connections. When a connection has been established, data can travel between the client and the server in either direction. When the data transmission is done, there is an option to close the connection. Some implementations keep connections alive for as long as possible, while other implementations close the connection immediately after each individual transaction. This all depends on what characteristics the data flow has. If small transactions are made continuously during the usage of a product, then keeping connections open might be the most efficient way of handling the problem. If data transactions are made rather seldom, then closing the connection after each transaction will most likely be the way that gives the highest performance. Another important thing to consider is

the use of push technology. If push technology is to be used, then an open connection is required. See **section 2.8** for more details.

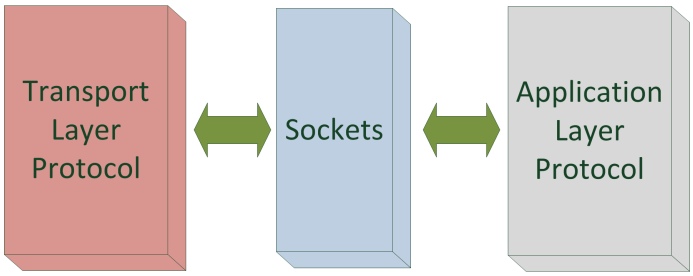

Figure 4: The sockets are the glue between the transport layer protocol and the application layer protocol.

Another important point with closing the connections are that there might be upper limits for how many connections can be open at the same time. This limit depends on programming language of choice, system architecture and operating systems and is thus hard to give a straight number on. This limit will be described in more detail in the implementation section.

In this project, sockets will be used as the backbone of the network architecture and will not be used directly by the game developers. As already mentioned, there will first be an application layer protocol that creates an abstraction of the problem. A User API will then further abstract the problem with commands that are very close to the real world.

2.6 Concurrency Model

Up until now the analysis section has, apart from the requirements section, consisted of an in-depth description of the protocol stack that will be used for the server design and implementation. Now, let's look at something completely different; the server's concurrency model. A well thought through concurrency model is of utmost importance for server performance. Without such a model, the server will be in great risk of choking under the pressure of thousands of concurrent clients.

Concurrency is the real strength of Erlang. No other language can compete with Erlang when it comes to running concurrent threads or processes. Actually, there are similar languages out there, but at least the languages that occupy over 98% of the world's usage can not compete [14]. Erlang can achieve this goal by using LWP's (Light-Weight Processes) that comes with a memory usage overhead of only 300 words per process [16]. Additionally, context switching between Erlang processes are between 10 and 100 times faster than switching between processes in the operating system scheduler (a program written in C for example) [9]. This means that Erlang can create almost endless amounts of processes without running out of memory and there have been unofficial benchmarks that report running more than 20 million Erlang LWP's concurrently on a single machine. This concurrency structure can solve the famous C10K problem [3]

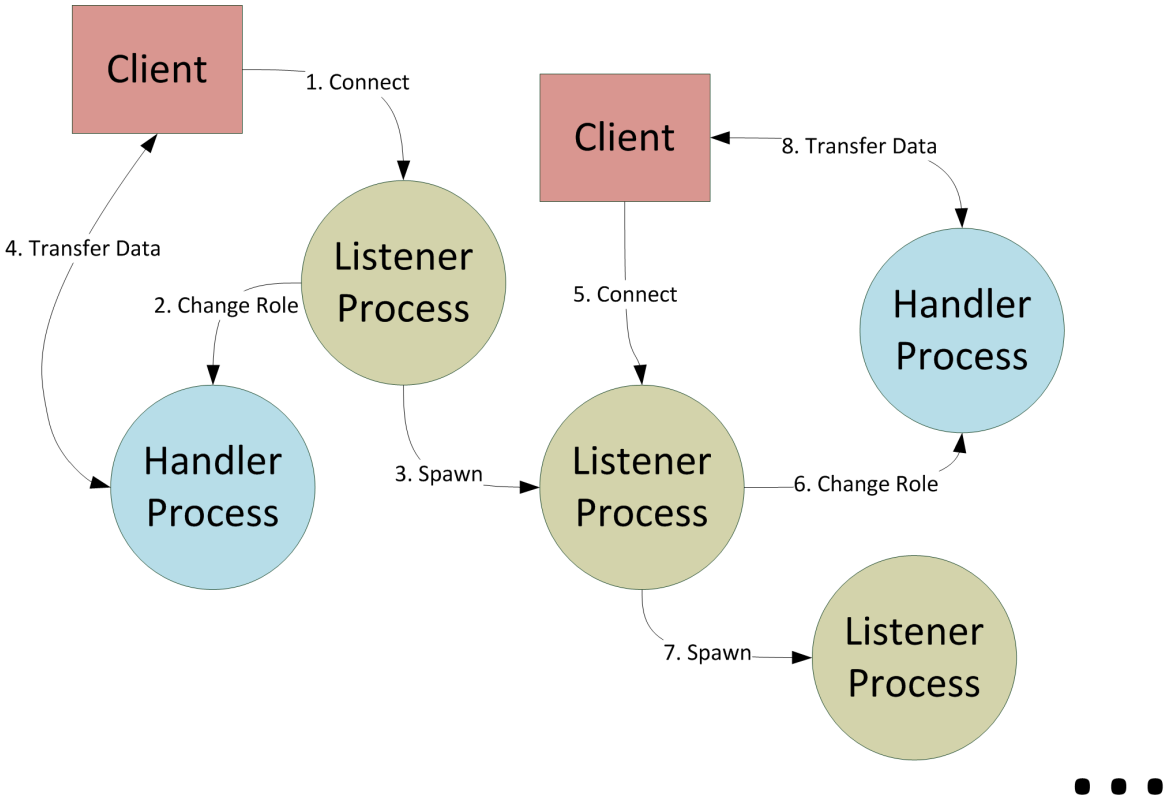

Figure 5: A concurrency model where every connection is handled by one process each. This is the natural concurrency model to work with in Erlang.

with relatively little effort put into the solution by the developer. C10K stands for Concurrent Ten Thousand Connection, and is a well defined problem for web servers. To solve the problem, a web server (running on a single unit) has to be able to handle ten thousand connections concurrently, as hinted in the problem's name. Knowing that there are websites that gets millions of visits each day, this problem might sound easy to solve. However, one must remember how the web works. Web servers does not necessarily keep connections open for any period longer than it takes for one request and one response to be completed. This means that a web server normally has an open connection with any particular client for a very short period of time (essentially every time the client does an action that requires a response from the server). Also, most large websites use server farms with multiple machines. Still, many popular websites need to handle immense number of clients and thus needs a server that can handle well beyond ten thousand concurrent connections in one machine.

Solving the C10K problem without relying mainly on processes and threads requires some inventive and difficult architectural choices. Looking at the world of web servers once again, Nginx is known for its very high performance and ability to serve high amounts of concurrent clients, well beyond ten thousand [12]. Nginx achieves this by using an event-driven architecture that does not rely on threads and processes. Apache Deft is another example, which uses only one single thread to handle any amount of connections [6]. More traditional web servers such as the Apache web server uses a con-

currency model based on threads and processes and relies on the principle of using one thread per client. The problem with relying on a single threaded form of architecture to handle massive amounts of clients is that it can get very messy, as the connections are not logically separated. This makes it harder to implement a fault tolerant system and it requires the developer to rethink the problem and design the solution in a, perhaps, not so logical way. On the other hand, the problem with using one thread or process per connection is the high memory usage, the operating system limit of threads and processes and the time it takes to do context switching. This is where Erlang really shines, completely ignoring the operating systems own implementation of threads and processes and instead using its own concurrency model within the Erlang VM. This also makes an Erlang server extremely portable, as the operating system the server runs on does not matter; Erlang can still create immense numbers of processes within its VM regardless of system architecture, as long as there are sufficient amounts of memory.

Yaws is an example of a high performance server written in Erlang and it makes use of the light-weight Erlang processes to achieve its goal. A comparison between the Apache web server and Yaws made in 2002 shows that Apache dies after 4000 active connections while Yaws is still functioning with over 80,000 connections [19]. Yaws explicitly states that it uses one process to handle exactly one client [15]. This makes the server work in a very human understandable and fault tolerant way, where connections are logically separated from each other and where the abstraction of the problem is relevant to the real world. If one process handling a connection terminates unexpectedly from a fault, no other connections will be affected and the systems state can easily be restored by restarting that single process.

2.6.1 Popular Concurrency Models

From the previous section, some different approaches to a working concurrency model can be derived. The natural way of handling concurrency in a server built with Erlang is to have each connection handled by an individual process. For other languages, there are a few different popular options. As already mentioned, Apache Deft uses a single thread to handle all connection and this is done by using non-blocking asynchronous I/O. Jim Petersson, co-founder of Apache Deft, states the most common concurrency models in his blog [22]: the *one client at a time approach*, the *threaded approach*, the *thread pool approach* and the *single threaded non-blocking asynchronous approach*. Here follows a brief description of those concepts:

- The one client at a time approach: this is the naive way of constructing a server. This hardly qualifies as a concurrency model at all. The idea is to simply queue incoming requests in a single thread, and to handle them in a synchronised order. This gives incredibly bad performance under heavy load, as only one request can be handled at each time. This is what is called a blocking server; the server is blocking when a client request is handled.

- The threaded approach: this is the approach that is most natural for an Erlang developer as one new thread or process is created for each new connection. This approach is, as mentioned earlier, not optimal in most languages, as each new thread and process comes with far greater overhead compared to Erlang and since context switching is very CPU demanding for most languages.

- The thread pool approach: similar to the last mentioned approach, with the difference that the threads are created when the server is started and they don't terminate after closing a connection. This gets rid of the overhead that comes with constant thread creation and termination. According to Jim Petersson, this is the most commonly used technique in web servers today. This approach does however limit the servers scalability, as the server is always allocating memory for the threads and since the server puts a hard limit of how many clients can be connected.

- Single threaded non-blocking asynchronous approach: the server runs only on a single thread, and instead uses non-blocking I/O libraries to create an event handling architecture for handling multiple clients simultaneously. This reduces the overhead compared to multi-threaded approaches and scales just as good as the threaded approach. What's not so good is that it requires the developer to remodel the problem to fit with the event handling architecture and this might not be as natural as working with the threaded approach. Error-handling also becomes much harder since the entire server is in risk as soon as an error occurs, as everything is handled by a single thread.

In conclusion, the threaded approach where a new process is created for each new connection and where the process is terminated at the same time as the connection is the natural choice for Erlang. First of all, it's logical to implement with Erlang's syntax and principles. Secondly, the Erlang VM and the light-weight processes are all highly optimized for such an approach. Thirdly, it makes the server as scalable as possible; the server uses exactly the resources required under any level of load and it can handle any steep load increase. In the design section, this concept is described in more detail.

2.7 Push or Pull

The protocol stack is defined by analysing the characteristics of the client/server-system's traffic. Parameters such as packet-size, time between transaction and data importance are considered. This has so far led to the choice of TCP and JSON for example. A question that remains is **how** the packets will be sent between the clients and the server.

The game server's basic functionality is structured in the way that:

1. A client sends data to the server.

2. The server calculates a new state based on this data.

3. The server sends the new state to all clients in the same channel (game) as the client sending data in step 1.

This data transaction can be implemented in various ways. Two popular models to rely on are Push technology or Pull technology.

In Push technology, data transfer is initiated by the server. The server **pushes** (or publishes) the data out to the clients [18]. A problem to overcome with this concept is how the connection is initiated. In the real world it's hard for clients to initiate

connections to a server because of firewalls; most firewalls blocks inbound traffic. This problem can usually be solved when using a fixed broadband connection by configuring the firewall in such a way that it can accept inbound traffic on specific ports. This solution is however not very practical for the end users and most end users might have no idea how to change their firewall settings. It also makes it impossible to play from many workplaces, for example. In the mobile world, where users communicate over the 3G and 4G networks, the firewall problem is commonly not solvable at all, as most big ISPs keep their entire mobile networks behind firewalls. This means that there is no way for the users to reconfigure the firewall themselves. What this means is that the client still has to initiate the connection even if the server is the one initiating most of the data transactions. Thus, the client has to keep a persistent TCP connection open at all times to allow the server to push data whenever it's needed. This is costly for the server, since there is an upper limit of how many concurrent connections the server can handle and since sockets consume memory. For the game server this would mean that each client has to keep a connection open all the time, even if data is sent very seldom.

Pull technology is the complete opposite of Push technology and relies on the concept of client initiated data transactions; the client *pulls* the data from the server [18]. Pull technology is closely related to Polling, where a resource continuously asks another resource if there is data ready to be transferred. For the game server it means that the server won't be able to instantly send an updated game state to all clients as soon as the data is ready. Instead, the clients have to initiate the connection and ask the server for the data. If the data is ready, the server delivers the data and if there is no new data, the server notifies the client of this circumstance. The connection is commonly terminated after any of these outcomes. Pull technology is costly as it constantly consumes bandwidth. For the game server this would mean that the clients must be configured to regularly poll the server for new data. The higher the polling frequency is, the closer it gets to behaving like a Push system. If polls are sent several times each second, the data transfer will seem almost instantaneous between clients. But when the frequency gets higher, the cost in bandwidth and CPU gets higher as well.

The choice between push or pull technology depends on the traffic's behaviour. As established earlier in the analysis, the traffic between the server and a single user will consist of small packages sent with large intervals in between; sometimes at the scale of 30 seconds while the user is active, and sometimes at the scale of several hours or days while the user is inactive. One thing that the server must support is to be able to notify users when new data is available. The user should not have to manually start the game up to check if something has happened. This means that if the user goes inactive for several days, and the game would poll for updates every 10 seconds, pulling technology will require enormous amounts of wasted bandwidth. This leads to the choice of push technology, as it's the logical choice for services where the application has a demand for instant updates even when the updates are far apart. Another positive thing about using push technology is that it's very straightforward to implement; just keep an open connection between the server and the client and send data as soon as the data is ready. Having the server constantly swapping sockets in and out required a bit more thought. The constant opening and closing of sockets also makes pull technology consume more CPU cycles compared to push technology where the

sockets are always open. The only downside with push technology, performance wise, is that more memory must be allocated to keep the sockets alive. Other services where push technology is used are instant messaging services, chat services and services for monitoring various data like stocks and sports results.

2.8 Analysis Summary

In this section, the analysis is summarised by describing the protocol stack and the mobile implications.

2.8.1 Protocol Stack

The game server uses TCP as it's transport layer protocol of choice, since it provides important features such as guaranteed ordered delivery of data. The interface between the transport layer and the application layer is made up of TCP sockets. At the application layer, a new protocol will be implemented on top of the data interchange format JSON. This creates a situation where the format is supported by all major programming languages and the protocol is at the same time specified to do exactly what is needed without any unnecessary overhead. On top of the application layer, a User API is implemented to provide an easy working environment for the game developers. The User API's functions are analogous to the use cases specified for the client role in the requirements section.

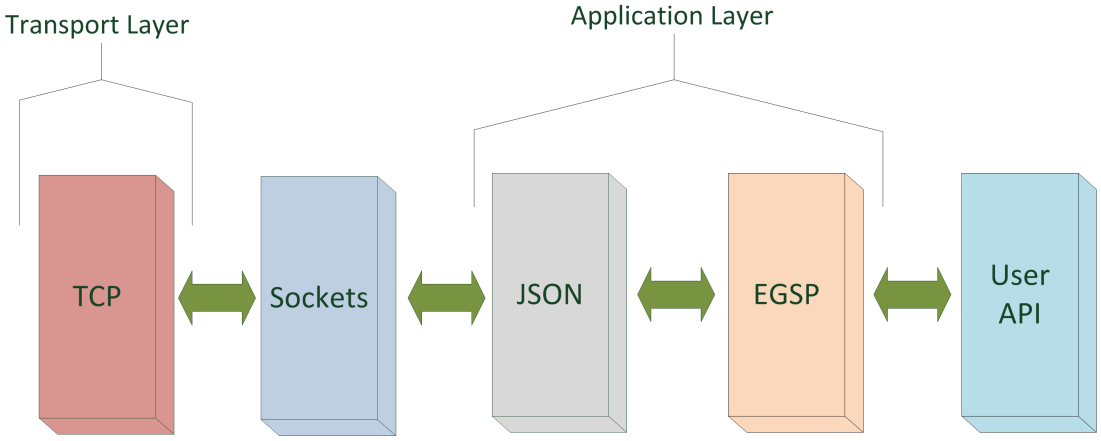

Figure 6: The game server's protocol stack. EGSP is short for Erlang Game Server Protocol.

2.8.2 Mobile Implications

The game server is constructed with mobile gaming in mind. This puts certain restrictions on the server functionality and also impacts some design choices. The server only supports turn-based games since the current 3G/4G technology is highly unstable in regards of RTT and loss and since turn-based games are what's currently popular to play on mobile phones. Adding support for real-time games is therefore a trade off; while some developers might want to use the server also for real-time games, others will find it unnecessary with additional complexity in the user API. To keep the API as easy to use as possible and to only deliver something that will work as intended, the

server will only support turn-based games. Other implications comes from the characteristics of the mobile traffic; again, fluctuating RTTs and high packet loss. This means that there has to be a well defined system for guaranteed deliver of data. This is provided by using the TCP protocol. When implementing the user API, correct language choices has to be made depending on the mobile market. The two big platforms are currently iOS and Android, who use Objective-C and Java respectively. C++ is another popular alternative for developing games on Android. Finally, mobile phones behave differently from PCs. A mobile phone can change IP often since a user might travel around and use different networks and WiFi-connections. This requires a system for the server to be notified regularly of a phone's current address.

3 Prototype 1

Before the design phase, a first simple prototype was constructed to do some initial testing. This prototype included four different modules: a Server Core, a Listener, a Connection Handler and a Dummy Client. The Core was constructed to be the backbone who sets up the environment and spawns a Listener process. The Core would then terminate itself, leaving only the Listener running. When the Dummy Client would attempt to connect to the server, the Listener would accept the connection, turn itself into a Connection Handler process and then spawn a new Listener process. This way the server can accept very large number of connections, with each process handling one client each. After establishing a connection, the Dummy Client could send data to the server and get an acknowledgement back from the server, letting the client know that the data was delivered intact.

A very interesting thing to test with this server was the number of connection that the server could handle. Initial tests showed that only 1024 concurrent connections could be established. After reaching this threshold, the server stopped accepting connections but it did not cause any faults. This limit was found out to be set by the Erlang environment variable *ERL_MAX_PORTS*. This variable was changed to 50,000. After this change another test was conducted by:

1. A client opening a connection.
2. Sending a chunk of data to the server.
3. Receiving a chunk of data back from the server.
4. Repeating step 1, 2 and 3 50,000 times.

This resulted in around 16,000 concurrent connections. The server refused to open any new connections after reaching this threshold but it did not affect any server operations concerning the first 16,000 clients. The server keeps working as expected and all connections are closed down without any faults when all 16,000 clients disconnect at the same time.

The server's hard limit of 16,000 concurrent connections is caused by various hardware and operating system limits. The prototype was tested on an average consumer laptop running Windows 7; this is not optimal for testing server software. However, reaching 16,000 concurrent connections is a very good thing, as it does break the limit of 10,000 connections specified as the C10K problem that was mentioned in **section 2.6**. This also shows that using push technology is a very realistic idea and that the server has the prerequisites to compete with other server solutions, or even beat them.

4 Design

4.1 User API

An important feature of the game server is a well defined and easy-to-use user API. The more powerful the API is, the easier it will be for other developers to make use of the game server. The user API implements all functionality described in the functional requirements section with some additional settings that allows developers to tweak the server according to their specific needs. As previously mentioned, an API is something that needs to be implemented in each separate supported language and the implementation can vary depending on language preferences. For example, some languages might want to use asynchronous I/O when waiting for updates from the server while others might want to block a separate thread (or block the entire application, if the game developer so desires).

Here follows a description of the user API; that is the API that is used on the client side to communicate with the server. To clarify: a *game session* in this text is a game that has not started yet and a *game* is a game that is in progress and consequently is communicating with the game mechanics module.

- `connect(Host,Port)`
  Connects to a specific server. Returns host IP if successful, otherwise error.
- `connectMany([Host,Port],[Host,Port],...)`
  Connects to several servers in a row. The first available server accepts the connection. Returns host IP if successful, otherwise error.
- `createAccount(userID,pass,email)`
  Issues a request to create a new account. Returns true if successful, error otherwise.
- `LogIn(userID,pass)`
  Logs a player into the system. Returns true if successful, error otherwise. Player must be logged in to be able to perform any other operations below. The userID is sent to the server with every other request.
- `LogOut()`
  Logs a player out of the system. Returns true if successful, error otherwise.
- `getPlayerData(searchString)`
  Returns a list of all players matched by the searchString.
- `addFriend(userID)`
  Associates another player as a friend. Returns true if successful, error otherwise.
- `removeFriend(userID)`
  Removes the association with another player. Returns true if successful, error otherwise.
- `sendMsg(msgString,userID)`
  Sends a message string to another player. Returns true if successful, error otherwise.

- `createGame(public,maxPlayers,timeout)`
  Creates a new empty game session. Returns the gameID if successful, error otherwise.

- `terminateGame(gameID)`
  Terminates a game session. Only possible if the userID is master of the game. Returns true if successful, error otherwise.

- `getGameList(running,finished,accepting)`
  Returns a list of all active game sessions, by gameID, in the system. Three different boolean options for returning games in different states.

- `joinGame(gameID)`
  Join a game. Returns true if successful, error otherwise.

- `autoJoinGame()`
  Join any open game. Returns true if successful, error otherwise.

- `recSessionState()`
  Waits to receive the game session state. Receives true if the game started as expected, false if the game was terminated.

- `startGame(gameID)`
  Start the game session. Returns true if successful, error otherwise.

- `invitePlayer(gameID,userID)`
  Sends an invite to a player to join a game. This can be used in combination with startGame(gameID) to create a challenge system. Returns true if successful, error otherwise.

- `getStatistics(userID)`
  Requests statistics about a user. Returns the statistics if successful, error otherwise.

- `makeMove(gameID,[attr1,attr2,...])`
  Makes a move. Only accepted if its the players turn. Returns true if successful, error otherwise.

- `recGameState(gameID,[attr1,attr2,...])`
  Receive new game state. Implemented in different ways depending on language. Might use asynchronous I/O in some cases, might be recommended to run in a different thread in other cases. Returns all data necessary to update the game state locally on the client.

- `kickPlayer(userID,gameID)`
  Kicks a player from a game. Only possible if the userID is master of the game. Returns true if successful, error otherwise.

- `setGameProperties(gameID,public,maxPlayers,timeout)`
  Change the game properties after a game session is already started. Only possible if the userID is master of the game. Returns true if successful, error otherwise.

Figure 7: Sequential diagram showing the typical usage of the server, by using the client API's functions. The green arrow notes that this call is derived from some other client updating the game state (by making a move).

4.2 Erlang Game Server Protocol

The application layer protocol is designed with the internal Erlang representation of each chunk of data in mind. When the server receives a chunk of data it parses its content with the EJSON parser and ends up with a list of tuples, where each tuple contains a key-value pair. This is a standard way of representing data in Erlang and it's supported by functions for searching, adding and removing pairs in Erlang's list-API. In the Erlang representation, each chunk of data begins with a tuple declaring what client the data is coming from. If the data is supposed to alter a game in any way, the id of the game is declared right after the client's id together with an atom declaring what action should be performed. If any more data is needed, that data is added in the third tuple. This is best illustrated by looking at some of the data chunks:

- `[{clientID,anders1398582},{gameID,game3948128},{gameData,x19y62w823}]`
  This is the data form for making a move in a game. first the client is identified, then the game and finally the game data is processed.

- `[{clientID,anders1398582},{joinGame,game3948128}]`
  Since joining a game does not require any additional custom data, the packet for joining a game need only be two tuples long with ids for the client and the game.

- `[{clientID,anders1398582},{newGame,game3948129}]`
  Creating a new game is represented in a similar fashion as joining a game.

- `[{clientID,anders1398582},{sendToClient,josef4958183}, {messageData, "hej hej"}]`
  Sending chat messages to other clients requires the remote clients id and the content of the message.

All commands presented in the User API section can be easily translated into this representation. For the Erlang Game Server Protocol, this data needs a representation in JSON. Note that if a client was to be written i Erlang instead of any other arbitrary language, the client could send data directly in the Erlang representation and ignore the JSON-structure. To best describe the data chunks in JSON form, the same four examples above are illustrated below:

Data chunk for making a move in a game:

```
{
    "clientID" :  "anders1398582"
    "gameID" :  "game3948128"
    "gameData" :  "x19y62w823"
}
```

Data chunk for joining a game:

```
{
    "clientID" :  "anders1398582"
    "joinGame" :  "game3948128"
}
```

Data chunk for creating a new game:

```
{
    "clientID" :  "anders1398582"
    "newGame" :  "game3948129"
}
```

Data chunk for sending a chat message:

```
{
    "clientID" :  "anders1398582"
    "sendToClient" :  "josef4958183"
    "messageData" :  "hej hej"
}
```

This simple structure greatly reduces the bandwidth usage in the server compared to using a protocol such as HTTP. It also makes the packages very human readable, which is good in the aspect of usability.

4.3 Architecture

The server architecture is divided into three subsections; the component decomposition, the module decomposition and the concurrency model.

4.3.1 Component Decomposition

The server is made up of three components; the Network Component, the Game Mechanics Component and the Statistics Component. On top of that there is a database that is used for storing statistics and for backing up the server state. As previously mentioned, this report only handles the server-side Network Module and therefore only that component will be described in detail. The module decomposition and the concurrency model sections does just that.

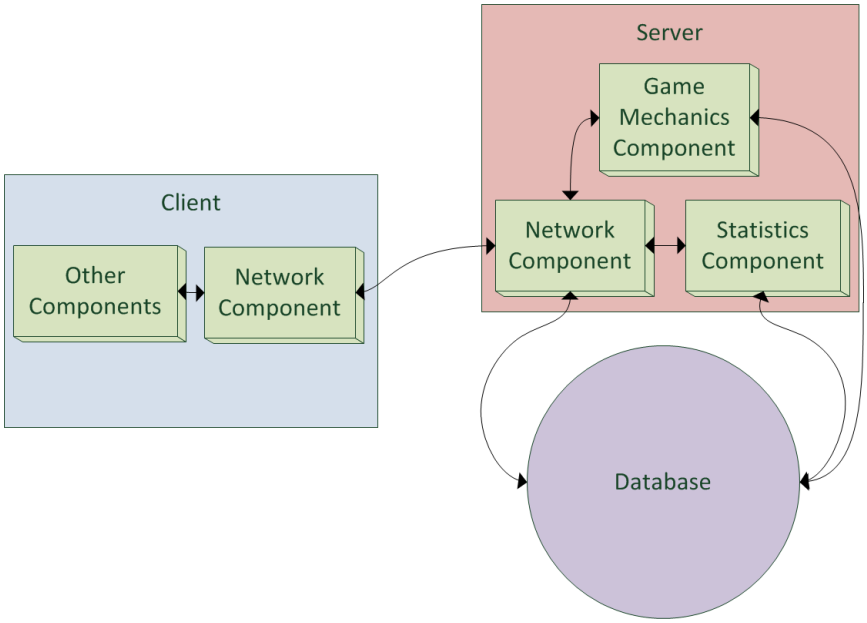

Figure 8: A simple view of the component decomposition. Since only the network part is included in this report, this is to be viewed as an intended component decomposition.

The Game Mechanics Component's main responsibility is to calculate a new game state based upon the data it receives from the Network Component and to send the processed data back. The data is passed through a Game Mechanics Interface implemented in the Network Component. The updated game state is also backed up in a database and can be requested when needed (if a game instance terminates unexpectedly for example).

The Statistics Component collects statistics of the traffic and operation of the server. A Statistics Interface is used in the Network Component for passing the data.

The network Component is responsible for handling all communication between all clients and the server. It's also responsible for keeping track of game states. This component's module decomposition is described in the next section.

4.3.2 Module Decomposition

The modules designed in this project handles the server's primary functionality; to be able to maintain a large world of players and game sessions. To achieve this, a number of different modules work together to create a logical and optimized structure. As previously mentioned, this project's only aim is to complete the network component of the server. Other parts like the handling of the database, the game mechanics and the statistics are only represented by simple interfaces that are up for future implementation.

Here follows a description of each module. For the formal definition of each module, see Figure 9.

- `Core`
  The core is the module that spawns all the essential server processes. The core takes input from the Internal Server Input Handler. The core also works as a monitor for the essential server processes and handles their exit codes.

- `Internal Server Input Handler`
  The Internal Server Input Handler works as an administrator portal for the server administrator. It provides the administrator with an API for handling the server, such as starting the server, adding games, removing clients and much else. See Figure 9 for the full list of commands.

- `Listener`
  The listener has only two functions; to listen to incoming connection calls and to accept the appropriate ones. Connections are either accepted, if the connection attempt uses the correct credentials, or denied. When a new client is accepted, the Client Controller is informed of this and a Client Handler is spawned for that client.

- `Client Controller`
  The Client Controller keeps track of all clients currently connected to the system. It has functions for adding new clients, removing clients and for checking if a client is already registered in the system. The clients are identified by their

ClientID and their Socket. The Client Controller also monitors all Client Handler processes and handles their exit codes.

- `Client Handler`
  A process running the Client Handler code is spawned for each new client that connects to the server. That process handles the socket connected to that client. The process receives and sends all data between the server and that client. If the connection is terminated, the process exits.

- `Game Controller`
  The Game Controller works in a similar manner as the Client Controller, only that it handles game instances rather than clients. It's functions includes creating new games, removing games and adding and removing players in those games. The Game Controller spawns a process running the Game Handler code for each new game session. It also monitors all Game Handle processes and handles their exit codes.

- `Game Handler`
  For each active game in the server there is a Game Handler process that keeps track of it. Among other things, The Game Handler process keeps track of the game state, what clients are connected to the game and whose turn it is. When the game is terminated, the process is also terminated.

- `EGSP Parser`
  The EGSP Parser module is used by the Client Handler processes and it parses incoming and outgoing data. After sorting and parsing incoming data, the Commander module is called to take appropriate actions. The EGSP Parser also receives data from the Commander, packs it into the EGSP structure and sends it out to the clients.

- `Commander`
  The Commander performs any action that is requested by the EGSP Parser. For example, if a client wants to join a game, it first sends that request to its paired Client Handler process. The Client Handler sends the request to the EGSP Parser. The Parser then figures out what action is supposed to be done, and tells the Commander to perform that action. When the action has been performed, the Commander also performs a response action to the EGSP Praser. The Parser packs that response into the EGSP structure and passes it to the Client Handler that sends it back to the client. The client is informed weather the action was successful or not (and the reason for not being able to join in that case).

- `Game Mechanics Interface`
  The Game Mechanics Interface is used by Game Handler processes to process game data. It has function for updating and delivering game states.

- `Database Interface`
  The database interface handles communication with the database.

- `Statistics Interface`
  The statistics interface handles communication with the statistics component.

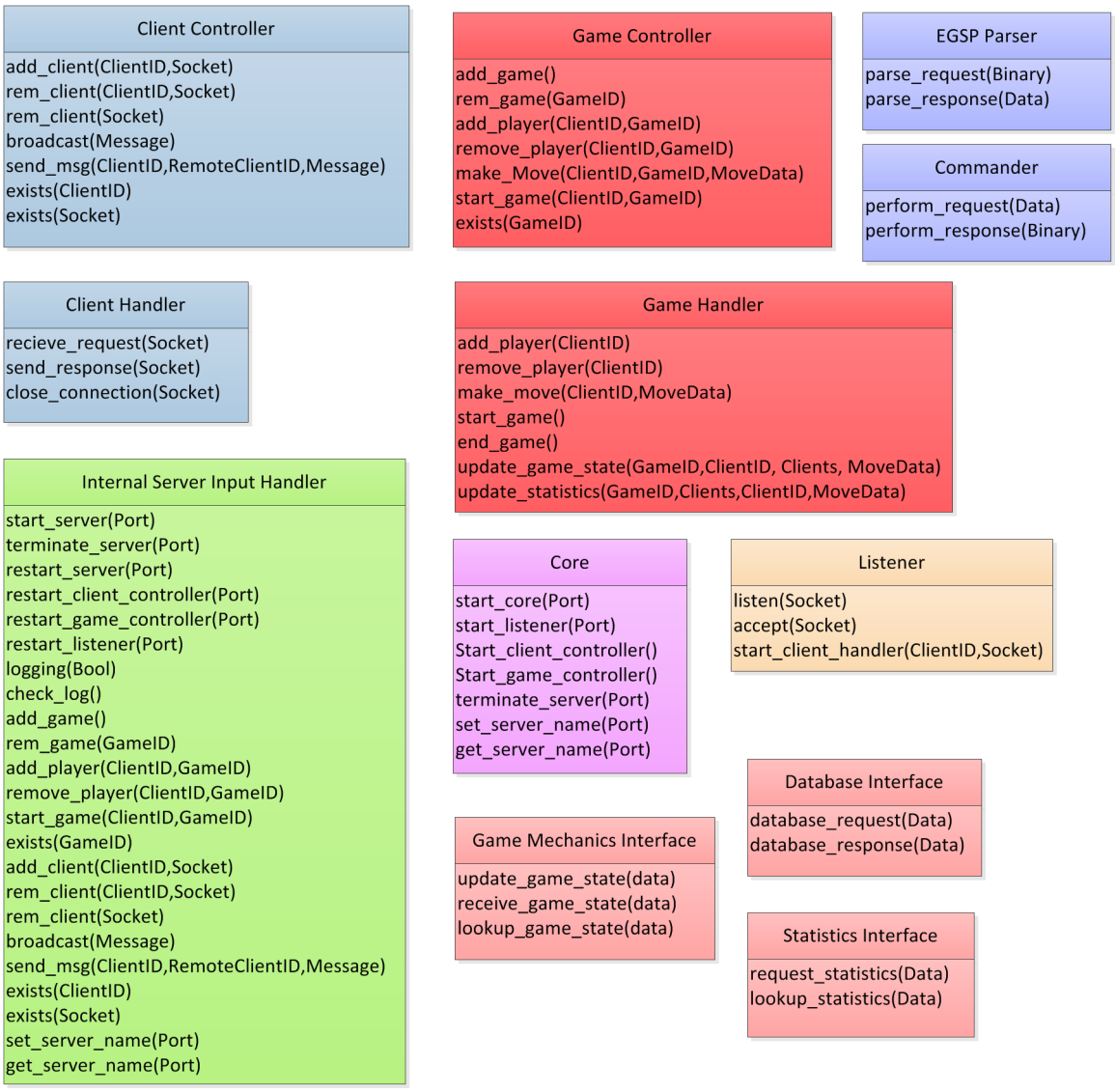

Figure 9: The module decomposition.

4.3.3 Concurrency Model

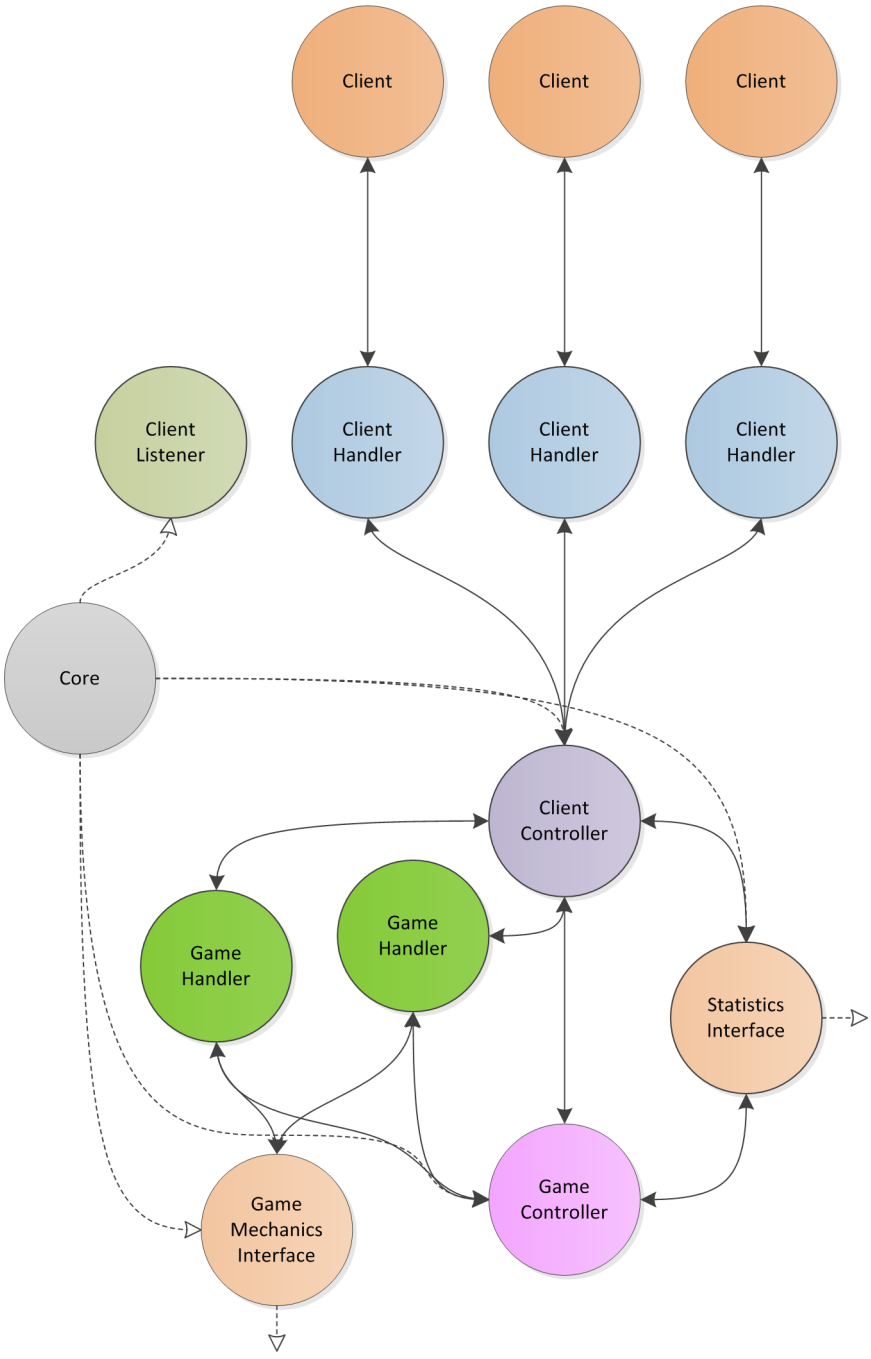

Figure 10: The server's process architecture at a state where three clients are connected to the server and two games are running.

The concurrency model works on the principle of separation of logic. The Core process is responsible for spawning the shell of the concurrency model, namely the Client Listener, the Client Controller, the Game Controller, the Game Mechanics Interface and the Statistics Interface. If any of those processes causes a fault and terminates, the Core receives their exit codes and proceeds with an appropriate action such as

re-spawning the terminated process.

The Client Listener process uses a listening socket and accepts incoming connections from clients. When a new connection is successfully accepted (or when a login attempt is made, from the client's point of view) the Client Listener process creates a new socket and passes it to the Client Controller, which spawns a new Client Handler and gives that Client Handler control over the new socket. One new Client Handler is spawned for each and every client connecting to the server. The Client Handler has several duties. It keeps track of its paired client's ID and matches this ID with incoming data, to guarantee that a client does not attempt to act as someone he is not. It also processes all incoming data from that client, and responds with a status message after processing each command. The Client Handler makes use of the EGSP-parser module to process incoming and outgoing data.

The Client Controller process keeps track of all clients connected to the server. When a new client is accepted by the Client Listener, that client's ID is paired with that clients Socket and is sent to the Client Controller. The Client Controller can then perform operations on the clients in the system. For example, if one client wants to send a chat message to another client, the communication is conducted by passing the message to the Client Controller, which then searches its data structures for that client's ID. Whenever a client is disconnected, it is removed from the Client Controller's data structures. If a Client Handler process is terminated unexpectedly, the Client Controller receives its exit code and proceeds with the appropriate actions such as re-spawning the process.

The Game Controller process keeps track of all the game sessions currently running on the server and performs actions on those sessions. When a client issues a request for creating a new game, the Game Controller is notified and spawns a Game Handler Process. This process keeps track of its game ID, what Game Mechanics Interface to use, what clients are in the game, if the game is started and whose turn it is. Whenever a client sends a request for changing the game session state (making a move or joining a game for example) the request is passed by the Game Controller and is directed to the correct Game Handler. The Game Handler passes the data to the Game Mechanics Interface that processes the data, changes the game state and sends the updated game state back. The game state is then broadcast to all clients connected to that session. When a game is terminated, the process ends and the Game Controller removes it from its data structures.

5 Prototype 2

After finishing the design documentation, the work on the second prototype began. This prototype is supposed to implement a large portion of the functional requirements while maintaining a stable and optimized environment. The list of usable functions to be implemented was decided to be:

- connect()
- LogIn()

- LogOut()
- sendMsg()
- createGame()
- terminateGame()
- joinGame()
- startGame()
- recSessionState()
- makeMove()
- recGameState()

With those actions being implemented, the server can be used in the real world to full extent. While some of the convenient functions (like addFriend() and getGameList()) are missing, the server will be fully functional internally; with clients connected to the server and with active game sessions.

The first thing to implement was the server Core, the Listener, the Client Handler and a Dummy client. With those parts in place, the server is similar to the server developed in prototype 1, as it can listen to connection attempts and accept them. This time around however, a different approach was taken in terms of how to spawn the Client Handlers. In the first prototype, the Listener process transformed itself into a Client Handler while spawning a new Listener. This time around the Listener process spawns a new Client Handler process and passes the socket to it. This was done for a more logical structure and it does not really impact performance as it performs the same actions. One should also note that the socket runs in blocking mode (or passive mode). The choice between blocking and non-blocking sockets does not make a huge difference in Erlang since each socket can be handled by a single process, meaning that the process doesn't have anything better to do than just wait for new data. This was chosen for better clarity.

After performing some tests and verifying that the server was running correctly, the next iteration began with implementing the EGSP Parser and the Commander module. This was a logical next iteration, since the server needs to be able to process data and commands before any other functionality can be tested. The parser was fitted with a main parsing function that uses pattern matching for extracting all of the commands mentioned earlier. The Commander was provided with code stubs at this stage, just being able to acknowledge that the Commands would be executed at the correct time. Some tests were then performed to test the input-space of those modules. A lot of bugs were actually found, and some of the code for parsing the packets had to be rewritten with stricter policies. Creating a waterproof input-space for a system that is open to any form of input from the Internet is not an easy task and this could probably provide enough material for an entire master's thesis on its own. With that said, the most common forms of "attacks" on the system will be blocked. A surprisingly large amount of time was put into this iteration, since testing the system for attacks is both

time consuming and a lot of fun.

The Client Controller was implemented in the next iteration. With this module in place, all stubs regarding client control in the Commander module could be implemented as well. The Client Controller was pretty straightforward since it mostly involves receiving different messages from the Commander module and to keep a persistent list of clients.

In the fourth iteration, the Game Controller and the Game Handler modules were implemented. The Game Controller could derive a lot of its characteristics from the Client Controller, since it uses very similar logics of receiving messages from the Commander and of storing a list of games. The Game Handler is however different from the Client Handler, since a Game Handler process needs to keep track of some persistent information such as players in the game, whose turn it is, if the game is started and so on. It also needs to be able to accept messages for adding players, removing players, executing player moves and so on.

In a part that can hardly be called an iteration, the three interfaces were implemented; the Game Mechanics, the Database and the Statistics Interface. Those interfaces are code stubs that acknowledge any input data.

Finally, the Internal Server Input Handler was created. This is a tool for the server administrator that provides a list of functions that can be used to interact with the server.

During each iteration, the dummy client was updated to be able to test that each new implementation worked as expected. After finishing the server implementation, a more sophisticated dummy client with parts of the User API described in **section 4.1** was made. A lot of tests were conducted with this client and it was a joy to see that the server did work as expected and that it could actually be used in the real world. A very simple game was even implemented as a Game Mechanics module and was played out between two players, with the server and each player all sitting on different machines. Figure 11 shows a sketch of what happens in the prototype when two clients are playing a game versus each other and when one player makes a move.

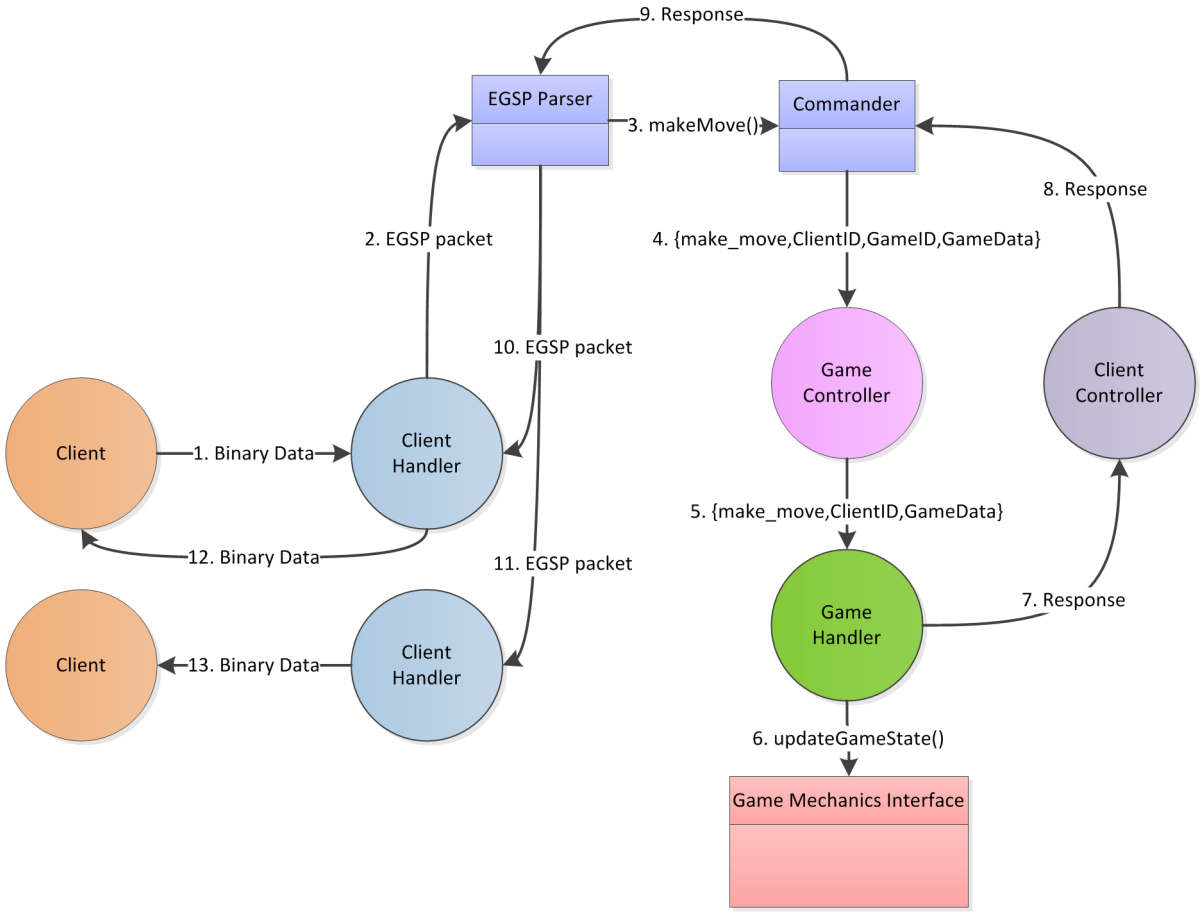

Figure 11: This is a simplified version of what happens internally in the prototype when a player makes a move in a game with two players.

6 Future Extensions

This section explores some future extensions that should be implemented to make the server more secure, more stable and to increase the performance.

6.1 Security

The major concern to be raised for using the server in a real world environment is security. It's obvious that there could be numerous entire master theses written on the subject of Internet security, since it's a very deep and time-consuming field. While the server filters its input in a sophisticated way and only accepts data from the correct users, some secure way of handling user data should be implemented. Storing user data such as usernames, passwords and e-mail addresses must be done very carefully or they might end up in the wrong hands. This could be done by using an external login-service, such as having users log in with their facebook account or OpenID. This would be a quick way of implementing a secure login system without having to do all the work from scratch. Another thing that must be implemented is some form of encryption. Currently, packets are sent as unencrypted binaries that could be caught by a man-in-the-middle attack or be sniffed up in a wireless network. Encryption of packets is not as critical as protecting user data since the risk is lower (the consequences for having an entire database of usernames and passwords hacked is much larger compared to getting some packets sniffed). Erlang includes a module called Crypo, where a crypto-server can be started and used to encrypt packages and data with a lot of different cryptographic functions such as MD4, MD5 and AES. The crypo-server can of course also be used for encrypting stored user data, if that would be the choice.

Another part of security within gaming is anti-cheating. This is a very important concept especially for multi-player games where a cheater can ruin the game for other players. Cheating can however be done in different ways depending on how the client is implemented. If all game logics are stored on the client side then the data can easily be modified and thus putting the problem outside of this project's scope. Note that this project aims to have all game mechanics implemented on the server, greatly reducing the risk of cheating. What can be done from the server side is to check for data consistency, keep stored game data safe and to possibly include a report system where users can report suspicious behaviour that can be used by future server administrators.

6.2 Scalability and Distribution

The current server architecture allows for very good scalability. This is achieved relatively easy when coding in Erlang, since the server can create just as many light-weight processes as it needs at any particular point in time. What has not been thoroughly designed in this document is a method for distributing the traffic load over several machines (a cluster). This is partly because there has been no hardware available for this to be implemented and tested. Solving this problem is however something that Erlang is very good at and the implementation of this should not pose a significantly large problem. Erlang comes with something called the Distributed Erlang System where a developer can seamlessly integrate several machines in a cluster of nodes [10]. Each machine in the cluster is assigned as a node and can then be connected to other

nodes (this works transitively as well). After connecting the nodes they can reach each other transparently via TCP sockets.

6.3 Fault Tolerance

The server has one fault tolerance mechanism implemented, namely the use of monitors. For each process running in the system there is another process that monitors that process and receives its exit codes. This means that if for example a client handler causes a fault, the client controller process can act on that fault and choose to restart that process. A way of achieving better fault tolerance is to use a distributed solution. The Distributed Erlang System was mentioned in the last section as a tool for making the server system more scalable, but it can also be used for making the server more fault tolerant. In case a server running on an single machine goes down, the use of monitors is of no use since it can't prevent hardware failure. This can be helped by having other servers in the cluster notice that one machine has gone down (by using monitors between different nodes in the cluster) and take over that server's responsibilities. This requires that restore points are saved across several machines for the other machines to know what needs to be restored.

6.4 Documentation of User API

This report does provide some good documentation on how the server functions, but it does not contain details about the User API. This would definitely be required for it to be released to the customer market. The documentation of the User API needs to be extended and should also include examples of how to use it. There should be one document for each supported language.

6.5 Database, Statistics and Game Mechanics

These components have already been presented as something that is not included in this report and they are indeed something that is needed for the server to be fully functional in the future. A good idea would also be to implement a sample game with the server for other developers to learn from.

7 Conclusions

An important conclusion drawn by the writer of this report is that this has been a project where it's rather hard to get quantifiable results. If a project's aim is, for example, to optimize a certain part of a compiler, it might not be too hard to measure this and see if the project results in performance gains or not. In a project like this, where a completely new piece of software is analysed, designed and implemented, the benchmarking part turns out to be the size of another full thesis report. To begin with, not a single competitor (like SmartFoxServer or Steamworks) can give any straight number whatsoever of how many users their servers can support. The only answer to find is that it depends on a number of factors, such as your own network code, the server hardware, the operating system of choice and so on. This poses a problem, as one of the interesting parts of this project is to end up with a high performance server. The only way to measure this would be to implement the same game and network code on several different platforms and then perform various benchmarks. This is however not possible, as the workload of such a test would be way to big. And even if this was done, it's still hard to draw any conclusions, as the different implementations might vary a bit depending on the developers skill in using the different platforms.

What can be said, however, is that the server prototype does perform very well. Even on a consumer laptop it was shown in the prototype implementation that the server has the capability of breaking the C10K problem with ease. This is partly due to Erlang being a very efficient platform for this kind of task, but also because of a well designed concurrency model. The server's limit of concurrent clients must definitely be viewed as a success.

Other performance aspects to look for in a project like this is CPU, RAM and bandwidth consumption. Those areas also proved very hard to test, since no server hardware was available during the prototyping and since there is no reference data to compare with. What can be said is that 16,000 open sockets with some sample traffic occurring consumed around 3 gigs of RAM and did not throttle the 2.6 GHz dual core processor while in a steady state. The test with 16,000 clients was conducted using the local feedback loop, meaning that the clients were running on the same hardware as the server. This means that there was no test in bandwidth consumption. The bandwidth consumption should however be **very** optimized, considering the minimal application layer protocol design. Using the HTTP protocol, which is a popular choice for turn-based games, would have consumed numerous times more bandwidth. To again go back to the CPU and RAM consumption, one should note that there are lots of other consuming factors that were not implemented in the prototype and that were not part of this project. The game mechanics modules, the statistics gathering and the database connections would increase the required hardware for handling 16,000 connections greatly. The server is, with regards to breaking the C10K problem, most definitely on par with other solutions out there and in some cases most probably faster.

Scalability is something the server does very well. The memory consumption is directly linked to the number of clients and games to be handled and there is no real choke points to speak of. This is given by the concurrency model, where memory is allocated when a new client joins and is deallocated when a client leaves (the same

goes for game rooms). This is something that is easily achievable with Erlang and the chosen design paradigms.

A lot of effort has been put into the usability of the server. The application layer protocol is designed to be human readable and the user API is designed to include as few options and actions as possible while still allowing developers to implement what they desire. This aim, to make the user APIs as minimal as possible, is something that really distinguishes this project from its competitors. All previously mentioned competitors in the related works section delivers very advanced APIs with tons of functionality. This is a good selling point for this product, as it's aimed at small independent mobile game developers and since it's specially made for turn-based games.

Robustness should have been described in greater detail in this report. Unfortunately there was not enough time to design or implement a good fault-tolerant prototype with distribution and supervision. This is something that needs to be design and implemented for the final product to be a viable system. Security is another concept that should be explored in much greater detail before the final product is deployed.

As a last note, the idea of this project was not to get a final product that can show off with amazing performance and robustness. The idea was instead to make an analysis of the current state of the gaming server world and to analyse what needs to be done to enter and beat certain areas of that competition. This has been done by analysing every step of the network stack, by analysing the different choices in concurrency models and by weighing different design options against one another. The final conclusion is that this report makes a well thought-through analysis, and by making well motivated choices its design will be used as the core network model in a product that will hopefully be good enough to be released upon the market.

8 Summary

The independent game development scene is growing very rapidly and with the increase in mobile gaming, so is the turn-based game genre. Game developers are usually good at coding game mechanics and creating beautiful graphics, but might not be that good in terms of coding the network parts. This paper presents an easy-to-use, scalable, high-performance game networking server solution coded in Erlang. This server solution is aimed at small independent developers.

This paper takes the reader through an exhaustive analysis where the server requirements are set, where each part of the network stack is analysed and where the choice of concurrency model is explored. The report concludes that making a server for only turn-based games is a good idea, since there is a big market for it. The analysis then concludes that TCP will be used at the transport layer and that JSON will be used to create a custom made application layer protocol called EGSP (Erlang Game Server Protocol). The User API will be implemented in Java, Objective-C and C++ to begin with, to support the largest platforms Android and iOS.

After the analysis phase, a first prototype is implemented to test if the analysis conclusions are valid. The prototype is a simple server that listens for connections and sets up accepted connection in separate processes. The prototype works very well and can handle up to 16,000 connections simultaneously.

The design section includes the design of the User API, the EGSP and the servers architecture. Only the network part of the server is designed in detail while parts such as the database, statistics and game mechanics are left for the future. The design splits the network part into 12 different modules that work together to provide all necessary functionality of the server. The design section is concluded by describing the concurrency model that will be used.

To test the theory behind the analysis and the design, another prototype is implemented; this time with more functionality, such as the ability for clients to login to the server, send messages to other clients and to create and join games. The prototype turns out to be very stable and can also handle up to 16,000 connections simultaneously like the first prototype.

The server will need some additional functionality to be a viable for real world use. More work has to be put into security, scalability, distribution and fault tolerance. The user API also needs better documentation to allow for higher usability. The goals that the project set out to achieve were reached for the most part. The second prototype worked, the design seems to allow for a high performance server and the user API's simplicity sets the server apart from the competition.

References

[1] An industry shows its growing value - BusinessWeek. http://www.businessweek.com/stories/2006-05-12/an-industry-shows-its-growing-valuebusinessweek-business-news-stock-market-and-financial-advice, 2006.

[2] Mobile games. app store strategies, business models & forecasts 2010-2015. http://www.juniperresearch.com/reports/mobile_games, 2010.

[3] The C10K problem. http://www.kegel.com/c10k.html 2011.

[4] Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent. http://www.gartner.com/it/page.jsp?id=1848514, 2011.

[5] Apache MINA - welcome to apache MINA project! http://mina.apache.org/, 2012.

[6] DeftProposal - incubator wiki. http://wiki.apache.org/incubator/DeftProposal, 2012.

[7] Documentation - erlang programming language. http://www.erlang.org/documentation/, 2012.

[8] ejson. https://github.com/davisp/ejson, 2012.

[9] Erlang – gen_tcp. http://www.erlang.org/doc/man/gen_tcp.html, 2012.

[10] Erlang/OTP R15B01. http://www.erlang.org/doc/, 2012.

[11] JSON. http://www.json.org/, 2012.

[12] NGINX. http://wiki.nginx.org/Main, 2012.

[13] SmartFoxServer: massive multiplayer game server for flash, unity 3D, iPhone/iPad and android games, MMO, virtual worlds and communities. http://www.smartfoxserver.com/, 2012.

[14] TIOBE software: Tiobe index. http://www.tiobe.com/index.php/ content/paperinfo/tpci/index.html, 2012.

[15] Yaws. http://yaws.hyber.org/, 2012.

[16] Francesco Cesarini and Simon Thompson. *Erlang programming*. O'Reilly Media, Inc., June 2009.

[17] Vaddadi P. Chandu, Karandeep Singh, and Magued Iskander. Innovative techniques in instruction technology, e-learning, e-assessment, and education. pages 241–248. Springer Netherlands, 2008.

[18] Michael Franklin and Stan Zdonik. Data in your face: push technology in perspective. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, page 516–519, New York, NY, USA, 1998. ACM.

[19] Ali Ghodsi. Apache vs yaws. http://www.sics.se/~joe/apachevsyaws.html, 2002.

[20] Keon Jang, Mongnam Han, Soohyun Cho, Hyung-Keun Ryu, Jaehwa Lee, Yeongseok Lee, and Sue B. Moon. 3G and 3.5G wireless network performance measured from moving cars and high-speed trains. In *Proceedings of the 1st ACM workshop on Mobile internet through cellular networks*, MICNET '09, page 19–24, New York, NY, USA, 2009. ACM.

[21] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson Education, Limited, March 2012.

[22] Jim Petersson. Hur löser vi C10K-problemet? | diversify. http://www.diversify.se/blogg/?p=157, 2010.

[23] C. Serrano, B. Garriga, J. Velasco, J. Urbano, S. Tenorio, and M. Sierra. Latency in broad-band mobile networks. In *Vehicular Technology Conference, 2009. VTC Spring 2009. IEEE 69th*, pages 1 –7, April 2009.

[24] Ian Sommerville. *Software Engineering*. Pearson Education, 2007.

[25] Randall Stewart, Michael Tüxen, and Peter Lei. SCTP: what is it, and how to use it?, 2008.

[26] Emeric Thoa. Money and the app store: a few figures that might help an indie developer. http://thegamebakers.com/money-and-the-app-store-a-few-figures-that-might-help-an-indie-developer.html, 2011.