Ville Sillanpää

Implementation of Backend Infrastructure for Social Mobile Game

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Program in Information Technology

Thesis

3 April 2015

Metropolia

Author(s) Title Number of Pages Date	Ville Sillanpää Implementation of Backend Infrastructure for Social Mobile Game 44 pages 3 April 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer Antti Koivumäki, Principal Lecturer

The goal of this thesis was to create a scalable backend infrastructure for a social mobile game, Last Planets, deployed in the Microsoft Azure cloud environment. The main part of the study consists of three sections: the first one describes the needs and considerations for communication middleware, the second one goes through the design principles and typical components in a game server architecture, and the third one explains how the aforementioned theory is put in practice in Last Planets.

As a result of the research, Photon, a networking technology developed by ExitGames, has been chosen to perform the role of a networking middleware, for reasons such as having a comprehensive set of features, good documentation and support, ease of use, and compatibility with all major mobile platforms. Furthermore, server architecture—designed to be scalable, responsive, fault tolerant, and well monitored—has been implemented on top of Photon.

The knowledge gathered in this study can be used as a guideline or reference when developing a backend infrastructure for a social game. Building a backend infrastructure can be a complicated task—depending on the requirements for the particular game—and this thesis was written to present a solid foundation to build upon. However, some major areas such as security and issues related to running game logic on the server-side have been purposefully left out of the study.

Keywords	Game development, social game, mobile game, network communication, scalable backend, server architecture

Metropolia

Contents

List of Abbreviations

IP	Internet Protocol, the main communication protocol for transporting datagrams across network boundaries
MMO	Massively multiplayer online, a term defining types of games where players can see and interact with vast number of other players simultaneously
MTU	Maximum transmission unit, a largest protocol data unit that the network can pass onwards
RTT	Round-trip time, the length of time for a packet to reach its destination and return back to the sender
TCP	Transmission control protocol, a core protocol of the internet protocol suite which provides reliable, ordered, and error-checked delivery of packets
UDP	User datagram protocol, a core protocol of the internet protocol suite which is a simple connectionless transmission model with minimum features

1 Introduction

This thesis presents a high-altitude view for communication and architecture level concepts for developing backend infrastructure for a social game. The goals were to utilize the information to successfully construct a backend infrastructure for Last Planets, a game project currently in development for Vulpine Games, while providing ideas for other developers to help them build their own backend infrastructures.

The thesis is divided into three main parts: the first one describes the needs and considerations for communication middleware, the second one goes through the principles of design and typical components in a game server architecture, and the third one explains how the aforementioned theory is put to practice in Last Planets.

The communication middleware section focuses on choosing a 3rd party solution for handling network communications. It presents important features to look out for and describes how to utilize them with concrete examples. Each of the sections aims to answer the following questions:

- What does the concept entail?
- Why is it used?
- Where is it used?

In some cases, ideas are presented on how the feature can be implemented—without going deep into the details because the thesis is not about implementing a networking middleware solution. The communication middleware is further divided into sections dealing with packet transmission, resource optimization, and issues related to working with a solution provided by 3rd party vendor. The packet transmission section explains what is needed for effectively relaying messages with different needs for reliability. The resource optimization section defines a variety of ways for optimizing the communication—and the possible tradeoffs that emerge when using them. The last section is there to present some additional considerations that a developer should be aware of when deciding to integrate a 3rd party networking solution into a project.

The server architecture section defines some typical components for building game server architecture, and it explains the fundamental principles on which a social online game can be built on.

The theory part is followed by a section that focuses on the implementation of Last Planets; it explains the choice for networking middleware in detail and describes the server architecture with information on how it is made scalable, persistent, fault tolerant, and well monitored.

This thesis is not meant to present all the theory behind the concepts but rather what is considered to be relevant for a developer to gain an overview of the necessary information needed for designing a backend infrastructure for a social game which uses a 3rd party networking middleware. Understanding the fundamentals is the most important part—the technical implementation can (and most likely will) change depending on the design criteria. Also, due to the large scale of the project not all of the aspects could be included in this thesis. Thereby, security and application level networking, such as latency compensation methods, are not presented here.

Reading this thesis does not require background in network programming because the concepts are not explained from the implementation perspective. However, a wide variety of games are being used as examples throughout this thesis, and thus, general knowledge of previous major multiplayer game titles will help to understand the examples.

2 Project Description

This thesis is done for Vulpine Games which is an independent Finnish game development studio focusing on mobile and tablet games. The company was established in June 2014 and since then has been focusing on its first major title, Last Planets.

A couple months into the development, Last Planets went through a major reformation. The current form of the game has been in development since September 2014. The development for backend infrastructure of Last Planets started in the following month and has been developed, so far, as a solo project by the author of this thesis.

2.1 Description of the Game

Last Planets falls into the social base defense genre, and it's built around a space theme (see Figure 1). Each player has been assigned with a planet; their main goal is to protect the planet and make its civilization strive for higher levels of development—while forming planetary alliances and gaining more influence in the galaxy.

Figure 1. A gameplay screenshot from an early development build of Last Planets.

The game is designed to be played globally which implies that its backend infrastructure needs to spread out on different continents. Deploying own datacenters around the

world is not a valid option for a startup with fairly limited amount of capital. Thereby, the game is built to be deployed in Microsoft Azure cloud computing platform. Since Last Planets is a competitive game, all of the important decisions related to gameplay need to be confirmed by the logic running in backend which makes the task of building a scalable solution a fairly complicated process. This thesis reviews concepts essential for building a highly scalable and robust backend for a social mobile game.

The business model for the game is free-to-play which means that the game can be played for free but if the player desires to make progress faster she needs to invest money in the game. Since this business model is supported best by providing the game as a service, meaning that development will not be discontinued after the launch, Last Planets is intended to be updated for a significant amount of time. Successfully supporting this model entails the server applications being capable of receiving updates without taking the whole service down.

In addition to being globally available competitive multiplayer game, Last Planet has features which are notably difficult to execute in a mobile multiplayer game: synchronized multiplayer sequences, such as the combat between players, building space stations co-operatively, and a persistent galaxy map where all players within the game can be found and interacted with.

2.2 Goals for the Backend Infrastructure

To be able to declare whether the project is successful or not, some measurable goals need to be defined for the backend infrastructure of Last Planets:

- Choose a communication middleware which meets all the requirements the game presents
- The whole backend architecture should be maintainable by one person as long as the number of servers are within reasonable limits
- Deploy updates to the server applications without taking down the service
- Automatically direct players to most optimal servers
- Prepare for worldwide deployment by being able to efficiently scale the service up and down and load-balance the traffic
- Validate all important player actions on server-side to prevent cheating

These are not all of the goals for the project but rather the goals related to the subjects presented in this thesis. Obviously, well-engineered software needs to be responsive, well documented, and secure, among other things, but these subjects are not considered in this thesis.

2.3 Tools

Last Planets is developed in the Windows environment; the main tools for programmers are Unity and Visual Studio 2013. The client side application is made using Unity and code is produced in Visual Studio 2013. There are some notable tools used inside Visual Studio 2013 which make the development process flow faster: ReSharper is used for code inspection, refactoring, and navigation; Autofac is used for injecting dependencies in classes; and NUnit is used as a unit-testing framework.

The server code is built on top of Photon Server SDK and networking communication on client side utilizes Photon .Net API. Reasons behind choosing Photon as networking middleware are reviewed in detail in the chapter concerning communication middleware in Last Planets.

Servers are deployed in the Microsoft Azure cloud computing platform. Multiple services provided by Azure are being utilized:

- virtual machines with Windows Server 2012 r2 operating systems are running the server applications
- azure table storage, a NoSQL datastore, is used for storing gameplay related data
- azure blob storage is used for storing things such as deployment files, configuration files, and assets
- azure traffic manager is used for geographic load-balancing
- azure cloud services are used for automatic scaling and round robin load-balancing

Besides Azure, there are other viable Cloud service providers such as Amazon and Google. Microsoft Azure is chosen for this project for the following reasons: it is highly recommended among Finnish game developers, and Vulpine Games is taking part in

Microsoft’s startup program called BizSpark which provides personal support and offers free monthly credits to be used in their service.

3 Communication Middleware

The first step in making multiplayer games is to get players to communicate with other players and servers. Understanding what is needed for this communication and the whole concept of network programming is a complex task, because it's difficult to learn "just part of it". There are many interrelated issues that without a gestalt understanding of the entire situation, it's hard to make sense of it. These communication level requirements are applicable from MMO games to social games, and to games with more limited number of players.

The purpose of this section is to provide information for choosing a proper 3rd party networking library for a multiplayer game. It is not focused on implementation but on describing relevant communication level elements for many multiplayer games; it also presents examples on where and how to use them. Even though this section is not concerned on how to implement these particular features, features presented here can be used as base requirements for programming own networking library.

3.1 Packet Transmission

There are many ways to affect the flow of packets in network transmission that can be utilized to gain desired performance characteristics. The first thing to consider is: which transport layer protocol to use? After explaining the differences between two most prevalent transport layer protocol, UDP and TCP, and concluding why UDP is usually the protocol to choose for real-time multiplayer games, the following needs are considered from perspective of networking library implemented using UDP protocol. These needs include the concept of virtual connection and automatic flow control.

Next, it is explained how to get the transmission running as smoothly as possible with the help of following features: multiple channel support, controlling maximum transfer unit, and automatic flow control. Later, traffic quality measurements and their practical implementations are explained. Finally, the last section explains why it is important to be able to simulate network conditions when developing real-time multiplayer game.

The concept of Nat punch-through (or network firewall traversal) is purposefully left out of this thesis since the focus is on developing a game where players connect to servers rather than other players.

3.1.1 Reliable and Unreliable Messaging

The basis of internet is a protocol family called TCP/IP. Its architecture is typically presented as a stack of protocol layers through which the information flows. There is no unified way of presenting the layer model. One that is often used is dividing included protocols into 4 distinct layers: application layer protocols, transport layer protocols, internet layer protocols, and link layer protocols (Javvin 2007, 4). Application level protocols, such as HTTP, are typically not used for real-time networked multiplayer games (except for emerging WebSocket protocol which is not reviewed here). Link layer and internet layer protocols are usually not concern for game developers. Thereby, for the purposes of this thesis only the transport layer is concerned.

The transport layer defines two main protocols for sending and receiving bytes called TCP (Transmission Control protocol) and UDP (User Datagram Protocol) (Albahari & Albahari 2012: 653). They are not the only ones out there but almost all activity inside the internet is utilizing these protocols.

TCP is an inherently reliable protocol which uses virtual connection, stream delivery, and flow control with support for full-duplex and multiplexed data transfer (Comer 2000, 209; Javvin 2007, 48). Some of the TCP's features are requirements for real-time multiplayer games such as creating a virtual connection and flow control; while on the other hand, there are properties on the protocol which make it unusable for certain real-time multiplayer games with low latency tolerance. These problems all stem from the way of how TCP handles lost and out of order packets: They are sent repeatedly until the receiver has received the packet, while sending of subsequent packets is stalled. This can increase delays to a point that is no longer acceptable in real-time networked multiplayer games. Some games also have imperative to be economic with their bandwidth consumption. Due to complexity of the TCP protocol, the size of the packet is larger than in UDP which consequently leads into higher bandwidth consumption. (Fiedler 1)

UDP is a much more simple protocol and it makes the transport layer considerably thinner. There is no connection between the communicating nodes and it does not guarantee reliable packet delivery, control the flow or order of the packets, and does not have fault tolerance. If there is a need for these features, the responsibility of implementing them is delegated to the application layer. While using UDP protocol means more work for the application programmer, control over the implementation enables a possibility to create communication middleware which meets all requirements of a networked multiplayer game. (Comer 2000, 198-199)

Important game events are necessary to be sent reliably to retain consensus about state of the game between players. When reliability is done on top of UDP, it can be implemented in a way that does not block subsequent packets, even if previous packets are not received successfully, unlike when using TCP protocol where dropped packets can lead into considerable delays. Usually this is achieved by monitoring the status of a sent packet: if packet is dropped or delivered out of order it is marked as dropped; after identifying the dropped packet, it can be reconstructed and sent again based on reliability needs. Another technique for dealing with possible congestions in network, caused by the reliability implementation, is supporting multiple message channels (this is explained in a Chapter 3.1.3). (Frohnmayer & Gift)

Besides reliable messaging, it can be useful to have support for unreliable but lower overhead communication, depending on the requirements for the particular game. Reason being that many games constantly synchronize the state of game objects. The state might include information like position and rotation values which, when received too late, are not usable anymore and thus are better to be sent unreliably while reducing the bandwidth usage.

An important thing to consider is that UDP and TCP traffic don't intersperse evenly on a connection. Since TCP and UDP are both built on top of IP protocol, underlying packets sent by each protocol will affect each other. Data will usually flow through the same node, from local area network to the World Wide Web, which usually creates a bottleneck. TCP's congestion control induces frequent packet loss on UDP connections, especially with multiple concurrent TCP connections having their congestion windows synchronized. (Hidenari etc; Allman & Paxson & Blanton 2009)

3.1.2 Virtual Connection

TCP is a connection oriented protocol which means it includes a concept of virtual connection. Virtual connection means that both sides of the connection are aware of each other: they know where to send packets and have knowledge of whether packets get successfully delivered to the other party. UDP does not have this feature but it can be implemented on application level if a need for constant two way connection arises. (Fiedler 3)

There are two reasons why having a virtual connection can be useful in games: both of the communicating sides can take initiatives and are aware of each other's online status without explicit communication. When both sides can take initiatives in delivering the data (the client does not need to poll the data constantly) the time and traffic needed for receiving the data can cut down to half. This kind of behavior is compulsory in games with low tolerance for network delays. For instance, a first person shooter game is dependent that the node acting as a server is able to forward information it receives from a single player to every other player participating in the game. Forwarding needs to happen immediately to keep players' views consistent and there's simply no time for player to request for the updated information. Keeping track if players are online can be useful in many scenarios. As an example, a game can have a chat room where players are able to see others who are online and reach them with messages.

Implementing a virtual connection over UDP is usually done in the following way: the client starts by sending a packet to the server—the IP address and port number of the server should be known to the client beforehand—with a header extended with ID representing the client. After the server has registered the client's ID along with the IP address and port number both parties begin exchanging packets at steady rate by either through normal communications traffic, or in the prolonged absence thereof, by so-called keep-alive packets. Keep-alive packets are usually empty UDP packets or packets with minimal non-intrusive content. Disconnection is determined when the time interval between packets grows beyond the preset timeout-limit. (Fiedler 3)

3.1.3 Multiple Channel Support

In many games, messages transferred over the network have different requirements in terms of reliability. As a basic example, a player can be receiving chat messages which

need to be sent reliably to ensure they are always matching with what is displayed for other players. On the other hand, the player is also receiving updates of constantly changing positions of objects in the game. If a reliably sent chat message blocks the communication channel momentarily update for object state might arrive too late. As a result, it would cause skewed view of state of the game and problems in interacting with those objects. Sending packets via different channels based on their priority and reliability needs is the way of dealing with these particular issues.

In the networking model of pioneering real-time multiplayer game, Tribes 2, which has been inspiration for many ensuing games in the genre, the transmitted data was organized in following categories:

- Non-guaranteed data is data that is never re-transmitted if lost.
- Guaranteed data is data that must be retransmitted if lost, and delivered to the client in the order it was sent.
- Most Recent State data is volatile data of which only the latest version is of interest.
- Guaranteed Quickest data is data that needs to be delivered in the quickest possible manner. (Frohnmayer & Gift)

Having the ability to distribute messages between multiple channels with different reliability characteristics is a requisite for making sophisticated system like the one present in Tribes 2.

3.1.4 Maximum Transfer Unit

MTU or maximum transfer unit refers to the maximum size cap for a packet sent to the network. IP packets can be as large as 64 Kbytes but there are stricter limits set by the link layer and different routers. For example, the MTU for packets traversing on Ethernet is 1500 bytes. When forwarding an IP packet larger than the link's MTU, IP interface must perform fragmenting – dividing the IP packet up into a sequence of smaller IP packets that all fit under the MTU limitation. Both UDP and TCP take responsibility of reassembling the fragments reliably which in UDP's case means that packet is dropped if not all of its pieces are received successfully. (Armitage etc. 2006, 58)

Being able to tune the MTU is a requirement for optimizing network performance. Making the MTU larger reduces the number of packets to be processed for same amount of data. However, smaller packets are more efficient in occupying a slower link which leads to smaller delays for subsequent packets, and decreasing lag and minimum latency. As an extreme example, a 14.4k modem is tied up for duration of one second when sending the largest packet allowed by the Ethernet. Large packets also induce more problems when communication errors are present. When single bit is corrupted the whole packet is discarded in UDP or resent in TCP. Larger packets are also more likely to corrupt and retransmission of such packet takes longer. (Murray etc. 2012)

3.1.5 Automatic Flow Control

Since UDP does not have automatic flow control it needs to be implemented on application level. What it means is that application dynamically scales the rate of data sent to suit the properties of the connection. If packets are being sent without flow control, one introduces the risk that the routers can't deliver messages at the same rate that they are getting dispatched into the network; messages will get buffered inside the router. This behavior can result in severe delays in communication. (Fiedler 2)

The most important metric to measure when trying to resolve if a connection is getting flooded is round trip time or RTT. In the simplest form, a flow control mechanism can be implemented by adjusting the amount of data to be sent according to the previous RTT values: if it takes a lot of time to perform a round trip, the amount of information should be reduced. (Fiedler 2)

3.1.6 Network Quality Measurements

This section covers three characteristics of the best effort IP service: latency, jitter, and packet loss as well as what kind of benefits, for multiplayer games, can be acquired by measuring them.

Latency refers to the time it takes for a packet to travel from its source to the destination. When talking about a two way trip for a packet, typically the term RTT or round trip time is used. Usually RTT is same is as latency multiplied by two but this is not always the case since some network paths exhibit asymmetric latencies meaning that the amount of latency is not same on both directions. (Armitage etc. 2006, 69)

Usually a ‘ping’ command is used to retrieve an approximate of RTT between source and destination. Ping operates by using Internet Control Message Protocol packets to examine the host and measure time it takes for the packets to travel back and forth on the network path. There is one important thing to take into consideration when using Ping: some routers handle Internet Control Message Protocol packets differently from other packets which can lead to different RTT times when comparing to actual TCP or UDP traffic along the same path. These variances are in the scale of milliseconds. (Armitage etc. 2006, 79)

Jitter denotes the variation in latency between subsequent packets. Jitter is usually defined mathematically by depending on the timescale over which the latency variation occurs and the direction in which it occurs. For example, a path can be showing an average latency of 80 ms with latencies bouncing from 60 ms to 100 ms for every other packet – this would be quite noticeable jitter in context of real-time multiplayer games, even though the long-term average latency is constant. (Armitage etc. 2006, 69)

Packet loss refers to a situation where the packet is lost somewhere along the path from source to destination in the network. The propensity of packet loss for particular path is usually described in terms of packet loss rate or alternatively with ratio of the number of packets lost per number of packets sent called packet loss probability. (Armitage etc. 2006, 69)

The information about the player’s latency can be applied into multitude of practical situations. Many methods used for latency compensation take the latency into the calculations. One of such methods is called Dead Reckoning. In Dead Reckoning, state of the object is predicted based on last received information. If the latency is resolved to be high, the algorithm would move the object further away from position indicated by the last received data. (Caldwell, 2000)

It is often desirable that players who match against each other are directed to servers which are located geographically in between the players to prevent unnecessary high latency. This can be done by measuring latencies from players to servers. Ping can also be used to detect players on a server who are hindering the experience for others by introducing too large latencies in the communication. When these players are identified appropriate procedures can be applied for dealing with them.

Ping defaults to using a small, 64-byte IP packet (Armitage etc, 2006, 79). Nevertheless, it is still worth considering the load it causes for the server when continuously examining the connection quality for many clients simultaneously.

3.1.7 Internet Simulation

Internet simulation means simulating undesired network behaviors such as latency, jitter, packet loss, and out of order packets. When developing complex multiplayer games, it is considered mandatory to be able to simulate demanding network conditions. The purpose of the simulation is to ease debugging all of the features implemented for dealing with demands from networking – all the way from communication middleware to prediction logic implemented in the actual game logic.

It is good practice to define the minimum viable network conditions under which the game should be playable. Naturally being able to simulate these particular conditions helps with making design decisions for game logic, as well as for the communication level implementation. By the time of writing this thesis, developing networked mobile games with synchronous interaction is particularly demanding because of the unstable characteristics of 3G-connections. Before launching the game into production, simulating the most extreme network conditions is crucial.

3.2 Resource Optimization

Networked multiplayer games have limited amount of resources at their disposal, and more than often, one needs to take these limitations into consideration when designing a networked game. Required resources correlate directly with the amount of network traffic a game generates, and can be concluded with the following *information principle equation*:

$$Resources = M \times H \times B \times T \times P \qquad \text{(Equation 1.)}$$

M is the number of messages transmitted.
H is the average number of destination nodes for each message.
B is the average amount of network bandwidth required for a message to each destination
T is the timeliness with which the network must deliver messages to each destination.
P is the number of processor cycles required to receive and process each message.

If the game is utilizing all of its available resources, increasing the expenditure on one of them means that other resource or resources need to compensate for it; or the quality of the gameplay becomes worse. For example, the aim is to increase the amount of data about the state of the game that is sent to players, one might need to decrease the number of players in a game instance to compensate for it. The following sections will present methods for reducing resource consumption, namely bandwidth consumption, by increasing processing. (Smed etc. 2006, 183 – 184)

The first thing that usually comes to mind, when talking about bandwidth optimization, is compressing the data. This feature is the only one handled completely by the communication middleware, and unfortunately it can only go so far. It is for this reason that the communication middleware should also have features that enable ways for a game application to cut down its bandwidth consumption.

3.2.1 Considerations about Optimization

Typically, in many games, such as action games with many moving objects or MMO games, there is more data to be sent than the communication channels can handle. In these cases it is pertinent that the communication middleware has features that allow the game to cut down bandwidth and manage its usage.

While many things can be done on communication middleware, many of the major optimizations are done purely on the application level, such as designing the whole game around the networking component and deciding which data is needed to pass through the network.

As mentioned before, networked multiplayer games are usually complicated systems, and it is not desirable to make the system any more complicated than it has to be. As with CPU optimization, it is not recommended to make optimizations without concrete data suggesting doing so. Multiplayer games are partly invisible systems (because of the not-so-easily monitored network component) that may appear to work but have lots of soft failures and inefficiencies. Optimization always obscures the original intent of the system and makes it more susceptible to errors.

Before doing any optimization it is important to understand the network limitations in the environment. Information about bandwidth capabilities for each player connected to the game can be used to deduct how much information can be sent, while retaining from congesting the connection and inducing delays in delivering time sensitive information. Usually the data what describes *what* is happening in the game needs to be delivered to each client in time to preserve consistency between clients. In many games though, there is additional information that is used to describe *why* things are happening in the game. While this information is not mandatory, it can be used to create more enjoyable experiences. The information about connection's bandwidth capabilities can be used to provide as much of this data as is possible. (Aldridge, 2011)

3.2.2 Data Compression

Messages can be compressed by utilizing more processing power when sending and receiving data. In data compression the process of transforming data into a format requiring fewer bits is called encoding, and reciprocally, transforming it back to the original format is called decoding. Compression methods can be divided into two categories (see Table 1). In lossless compression no information is lost during the transmission and after reconstruction both ends have the exact same information. Obviously higher compression rates can be achieved by employing lossy compression where data is filtered to what is considered relevant to the receiver. (Smed etc. 2006, 190)

Data compression techniques can be divided into internal and external categories on the basis of how compression is applied to message. In internal compression contents of message are optimized to be efficient and non-redundant with regards to its own information. For example, if a game does not need to use floating point for positioning elements and it is plausible to use 16-bit fixed point values, the values can be converted into the more bandwidth efficient format. (Hook)

Table 1. Compression technique categories

Compression	Lossless	Lossy
Internal	Encode the message in a more efficient format and eliminate redundancy within it.	Filter irrelevant information or reduce the detail of the transmitted information.
External	Avoid retransmitting information that is identical to that sent in the previous messages.	Avoid retransmitting information that is similar to that sent in the previous messages.

On the other hand, in external compression contents of a message are optimized by removing redundant information after comparing it with what have been previously sent. External compression has a requirement of reliable message delivery. Otherwise, it would not be possible to trust that client has the same reference points for which the information is compared to. The term for this type of compression is delta compression. A typical example of delta compression in multiplayer games is to send only the delta of player's movement information and withdraw from sending any information when player is staying still. (Smed etc. 2006, 190)

3.2.3 Automatic Message Packet Aggregation and Splitting

Message packet aggregation means that information from multiple messages is merged into a single message. Since every UDP packet has 22-byte private UDP header, which is there to tell every node on the way to its destination where it is going and how large is the packet, sending more messages in one packet preserves bandwidth. To illustrate this in the context of the Equation (1), the average message size (B) increases while number of messages (M) and timeliness (T) decrease, and the overall bandwidth consumption is reduced at slight costs in processing (P). The decrease in timeliness means that the game becomes less responsive. (Smed etc. 2006, 191)

There are two approaches of handling the decision when packet is ready to be sent to the receiver. In *quorum-based approach*, messages are usually aggregated into a packet until size of the packet is about 1400 bytes which is the “ideal” MTU size. Quorum-based method guarantees savings in bandwidth consumption but delays can grow longer than what is acceptable for the game. In *time-out based approach*, messages are gathered during fixed time period and sending is initiated upon time-out. In time-out approach the delays are predictable but bandwidth savings depend on the message initiation rate—in the worst case no bandwidth is saved. (Hook) (Smed etc. 2006, 191)

Usually the most optimal solution in context of networked multiplayer games is the combination of both approaches. That is to aggregate messages only up to the ideal MTU size while preventing delays from growing too long by sending messages on a set interval even if size of the packet has not yet grown to the optimum. (Hook) (Smed etc. 2006, 191)

Sometimes packets can also grow larger than the path to the destination can handle without splitting the packet. On these circumstances, it is convenient to provide packet fragmentation and reassembly services on the networking framework. These services make handling the data easier for the application utilizing it. One way to handle packet splitting is to have sub-sequence number inside the header which is there to indicate that packet is part of larger entity and what the placement of this specific packet is in the whole entity. (Hook)

3.2.4 Message Priority Level Support and Receiver Groups

For providing seamless multiplayer experiences in games, where there is more dynamic data available about game state than the bandwidth capabilities of network can handle, an option to prioritize messages is necessary. In message prioritization, messages are divided into categories based on their urgency to be delivered to the receiver. The data to be sent is chosen in priority order and capped to the amount that the network can handle.

The network model in a popular multiplayer game Halo Reach serves as a practical example where aggressive message prioritization is necessary for delivering consistent and responsive gameplay. A typical multiplayer scenario in Halo Reach involves 16 players and several thousands of dynamic objects which can be affected by the player

interactions. Exchanging all this data over the network is not feasible which is why there is complicated set of rules behind prioritizing the data to be sent to the clients:

- priorities are calculated separately per object and per client
- priorities are based on client's view and simulation state
- distance and direction are used as core metric for priorities
- size and speed of the object affect priorities. (Aldridge, 2011)

By combining these afore mentioned rules, Halo Reach is able to deliver seamless multiplayer experience even if there are way more information about the game world that the network bandwidth capabilities could handle. (Aldridge, 2011)

Receiver groups simply mean that the node sending a message to the end recipients is able to determine who are allowed to receive the message (who belongs to the receiver group). This is a particularly important feature for MMO type of games since there can be from hundreds to thousands players in a single area; not having ability to choose which players actually need to get particular information would lead into overloading the network with data that is actually redundant to the gameplay.

3.3 Additional Considerations for Using 3rd Party Networking Middleware

Usually when large game studios are building games they prefer using proprietary software. Even though it consumes considerable number of working hours to build a robust communication middleware, it can be a make or break kind of deal for the project. While in the best case scenario, developer gets all the support, documentation, and reliable people working on the middleware software, the situation might not be as optimal which puts the project under a heavy risk. For projects running on a lower budget there might be no other solution than using 3rd party networking solutions.

In addition to general risks of using 3rd party software, there are some considerations that are specific to networking middleware solutions; when choosing the solution for

doing the task, there is no silver bullet or single technology than can do everything better than all others.

The first question to ask is does one want to manage the servers by himself? Some networking middleware providers are offering to run the game on servers hosted by them. Usually when this is the case, the customizability of the server is limited considerably. Some solutions do not allow running custom code on the server-side at all which can limit the server solution considerably. For example, building an authoritative server and calculating physics on the server are usually out of the question. On the other hand, for some game developers, low maintenance servers are the most attractive option.

Many networked games—especially popular ones—are faced with security and server overload problems. Making changes to the protocol that is used to pass on the game data is to make it more effective and secure is common. A networking middleware should make it easy for game developers to make changes to the content protocol. (Hsiao & Yuan 2005: 50)

Nowadays it is typical to release a game in multiple platforms. Some middleware provides PC client libraries, whereas other middleware is aimed at game-console or mobile-device clients. It is important to take this into consideration when starting to develop the game since it will otherwise limit possibilities for porting the game to other platforms. (Hsiao & Yuan 2005: 53)

4 Server Architecture

There are some key issues which need to be considered when designing networked online games that are intended to facilitate considerable number of concurrent players, such as:

- Scalability
- Persistence
- Player to player interaction
- Fault tolerance
- Monitoring
- Security

Scalability concerns how to structure a game application that dynamically adapts to varying demands on server resources caused by fluctuation in number and locations of players online. On large scale this can be achieved only if network of nodes can be harnessed to perform asynchronous computation. When designing online games, the speed of the scaling—both upwards and downwards—is an important factor since number of players can be hard to predict, and it can go up and down rapidly—especially during the launch of the game.

The concept of persistence means sharing common game state between player and the game application and upholding it even when the player is not playing the game. First, the game application shares the game state with the player who is joining the game. After the player shares the common game state with the game application these two entities start living in a symbiosis where both are responsible for initiating changes in the game state.

Player to player interaction means that players can have an effect on each other's gameplay experience. To support this kind of interaction, the game needs to provide a player with rich, accurate, and current information about other participants. The choices in implementation of persistence, scalability, and choosing the communication architecture play an important part when creating a solid base for interaction in the game.

Fault tolerance means that the server architecture can function and repair itself in the presence of software and hardware failures and monitoring is added to the equation to ease automated and manual maintenance. Security concerns keeping sensitive information out of reach for the clients, defending against denial of service attacks, and preventing players from cheating in the game among other things. The topic of security is outside the scope of this thesis.

4.1 Communication Architectures

When it comes to communication architectures, there are three prevalent alternatives; each one having their strengths and weaknesses. These architectures are peer-to-peer architecture, client-server architecture, and server-network architecture which combines the previous two architectures.

In the peer-to-peer architecture all nodes participating in the same game instance are interconnected to each other by the network. All nodes share equal status, and thus, there are no hierarchies between them; every node can communicate with any other node in the network. Since every node is responsible for delivering its status to everyone in the network, peer-to-peer architecture has limited scaling potential. Peer-to-peer architecture has been popular choice in the early stages of history of networked games and it is still useful when the number of participants is small or they communicate in a LAN-environment. (Smed etc. 2006, 175-176)

In the client-server architecture one of the nodes is promoted to a higher position in the hierarchy and all communications is handled through this server node. Client nodes only need to send data to the server which will distribute it further to other clients. Although packets need to travel extra step through the server when traversing to recipients, there is a variety of ways in which server node can optimize the communication. The server can take control of the packet flow and distribute only the data that is actually relevant for the particular players. Furthermore, the server can aggregate packets and smooth out the packet flow. (Smed etc. 2006, 175-176)

The client-server architecture can be taken one step further: to the server-network architecture. The server-network architecture can be thought as combination of servers connected to each other, as in peer-to-peer or hierarchical network of servers, and cli-

ents connected to servers following the client-server architecture model. Clients connect to their local server which has connection to remote servers which further provide connection to remote clients. The server-network model decreases capacity requirements imposed on a single server and thus increases scalability potential, but at the same time, it makes handling the network traffic increasingly complex (Smed etc. 2006, 175-176). Since the server-network model is the most viable option for creating a networked game which has massive number of users, is secure, and enjoyable to play all around the world, the contents of the following chapters are most relevant for this particular architecture type.

4.2 Scalability

Scalability means that the game application has an ability to dynamically adapt to varying demands for resources. In a networked game, resources are typically communication capacity, computational power, and data storage size. To achieve scalability on a large scale, there must be hardware parallelism in the system; multiple nodes must be working together for the game world: performing computations and handling communications.

If there were only computational requirements for the system, it could be solved –to an extent—by increasing the resources on a single node, typically by increasing the number of CPUs and amount of memory on a single computer. On the other hand, it would make communications ineffective, due to lack of geographically distributed server nodes. This is often referred as vertical scaling. But, with networked games there is an inherent need for serial computation because players need to have interaction with each other. Scaling by adding more server instances is often referred as horizontal scaling. Thereby, the speedup is bound to be limited, and when designing networked games it is important to understand this limitation for scalability. Also, while adding more nodes to the network increases computational capabilities there is a price to pay in burden it lays on communication channels between the nodes. (Smed etc. 2006, p.187)

Load balancing is a key component in achieving high scalability in networked games. Load balancing means distributing the load caused by players across multiple server nodes. There are two methods for load balancing: filling up one server completely and

then continuing with the next available server, or distributing the entire load evenly from the beginning. There are certain issues that can surface when performing load balancing, such as how to move player state between the servers. One solution to the issue is making the servers as stateless as possible. Being stateless means that no information is stored about the player on the server handling the computations; player state is loaded from a separate data source each time player related operations are being performed. (Pålsson etc. 2011)

Understanding the server architecture's capacity to facilitate players is important when planning for number of servers needs to be deployed for particular number of players—and in estimating the costs for running the game. For the communication capabilities following condition must hold:

$$d \times f \times n \leq C \qquad \text{(Equation 2.)}$$

d is the number of bits in a message
f is the transmission frequency
n is the number of unicast connections
C is the maximum capacity of the communication channel

For example, by using the values d = 1000, f = 5, and C = 10 Mbit/s, one can solve the maximum number for players on the server. Thus, if a client–server architecture is being used, one server can provide serializability for 2000 players at most. For mobile games the update frequency of 5 transmissions per second is realistic, but for high-end games, it is usually higher and the payload much larger, and consequently, the estimation of number of players is highly optimistic. Moreover, increasing communication increases also computational requirements. (Smed etc. 2006, p.188)

4.3 Persistence

There are three different architectures for controlling the data, and there are two attributes, consistency and responsiveness, which define their characteristics. High consistency comes from high bandwidth and low latency between nodes controlling the data and overall small number of nodes sharing this responsibility. On the other hand, high responsiveness comes when nodes can respond quickly on queries made to the data. It implicates that nodes controlling the data need to be spread out closer to the nodes making the queries. Thus practically speaking the two attributes correlate heavi-

ly with each other and the final solution is always a tradeoff between the two. (Smed etc. 2006, 177-178)

The three architectures are centralized, replicated, and distributed data architecture. Centralized architecture is the one with highest consistency since the idea in it is that one node stores all of the data. Obviously, the price for high consistency needs to be paid in lack of responsiveness, which is a pertinent part of many networked games, hence making the centralized architecture not usually suitable for widely available MMO game but for social game with fewer requirements for timeliness it can be an option to consider. (Smed etc. 2006, 177-178)

In replicated data architecture a copy of the same data exists on multiple nodes. It provides higher responsiveness but causes lower consistency since the data needs to be synchronized constantly with other nodes holding the data. The assumption with this model is that all players have a need to query data that is containing information about any player connected to remote data controlling nodes, i.e., all players can interact with any of the other players participating on the game. (Smed etc. 2006, 177-178)

In distributed data architecture, as the name suggests, data is distributed on multiple remote nodes based on where is its most optimal location. Usually in multiplayer games, players are not able to interact with all of the players at any given moment. This naturally reveals information that can be used when distributing the data among the nodes controlling it. A typical example in MMO games is to divide (shard) the game world in a way that players from different geographic locations (typically divided by continents) are playing in different instances of the game. Distributing the data can lead to high responsiveness and even relatively high consistency if players are mostly querying the data located at their closest data node. (Smed etc. 2006, 177-178)

4.4 Fault Tolerance

Fault tolerance can be defined as: The ability of a system or component to continue normal operation despite the presence of hardware or software faults and the number of faults a system or component can withstand before normal operation is impaired.

It is impossible to create a system which is completely immune to faults; there will always be a risk of failure. Being fault tolerant means that server should be able to follow a given specification when some parts of the system fail. These specifications are system specific and the implementation varies case by case. In an optimal situation there is no single point in the system that can take the whole system down when it fails; every component in the architecture is designed to be redundant and is backed up with a replacement component which is desirable to be performed automatically and without causing any downtime for the service. (Gärtner 1999)

Game servers differ from many other types of servers in that high-availability is often desired over safety of the system. For example, when an error in a system would result in risking human lives it is better to let the system crash safely than proceeding with faulty behavior. While a game server should be kept running as long as there are no game breaking behaviors occurring. Players are contented having the game running rather than having to wait during server down-time.

4.5 Monitoring

Since networked game requires multiple participants, server nodes and player nodes, to function, monitoring the entire system can be a difficult to achieve. It is important to implement comprehensive monitoring from the beginning of the development process to ease spotting errors in the system. There are two main things to monitor from server-side: application logs and performance counters. In addition, there is a variety of ways to monitor networked games in the client side code as well but they are outside the scope of this thesis.

In the production environment logging is one of the only applicable ways to detect errors in a system. There are a few guidelines for creating effective logging in a networked game. First, log entries should be identifiable to the client whose interaction

with the server caused them. Without it, even with only tens of concurrent players, it would be extremely difficult to trace actions which lead to the error. Second, there should be a possibility to define levels for the log entries. In popular logging framework for .NET, called log4net, there are following log levels defined (from the highest level to the lowest):

- Debug
- Info
- Warning
- Error
- Fatal

By breaking down log entries to categories such as these, and with possibility to choose which ones actually get recorded for the particular build, it is possible to prevent creating unnecessarily large amount of log data, and prevent servers from slowing down because of too much processing power used for logging—while still retaining the logs that are relevant to the state of the development.

In addition to logging, it can be useful to monitor different counters telling information about state of the computer instances running the server applications. For example, for Windows Server there are many predefined performance counters that can be used to provide information on how well the operating system, application, and service is running on the computer instance. Some practical use cases for information from performance counters are automatic server reboots, firing alerts for administrators, and directing traffic to more stable servers.

In the case of having server architecture consisting of multiple nodes, it is key to have the monitoring system centralized in one location—for providing ease and speed of access when it is necessary to locate faults in the system. In action, it means that all of the servers are continuously uploading their logs and counters to the node responsible for gathering the data and presenting the data. Ideally, all of the data is easily queriable based on variables, such as server application type, client's id, timestamp, and server id.

4.6 Typical Components

There are many components typical to game server architectures which are not tied to any specific game genre. However, ultimately, the used components are game specific and the requirements are dictated by scalability and security requirements, communication model in use, and computational power demanded by different operations. Some typical components present in game server architectures are the following:

- Master server
- Proxy server
- Login server
- Asset server
- Gameplay logic server
- Memory cache
- Database

Dividing the functionality into smaller components enables the system to scale effectively on those areas where it is actually needed. For example, login service can take a considerable amount of computing power per client for short period of time (when the authentication is being processed); if it was implemented on the same server application as where the game logic resides, the whole application would need to be scaled up momentarily when the login service is under high-load; possibly resulting in decreases to quality of the service for game logic operations. On the flip side, there are also tradeoffs to dividing the system to smaller components: communication between server applications is more complicated than internal communication (inside single application), and thus leads into higher development times, and can be more prone to errors. It should be noted that these different components do not necessarily need to be separate physical instances but server applications. In fact, in the name of cost effectiveness, under low-load conditions many of the applications should be run on a single server instance to reduce expenditures from paying for multiple servers when the load does not actually require it. (Dieckmann 2013)

The master server is responsible for keeping track of responsive game servers and directing users to the most optimal server instances in terms of load and geographical

location (Bernier 2001). This is called load balancing. While load balancing can be handled manually, current cloud service providers provide automatic load balancing with customizable conditions as part of their service.

The proxy server can serve multiple purposes: increase performance of other servers, improve security, and make handling players more feasible. A proxy server can take load off from other servers by managing client connections, packet encryption, and packet compression. Security improves when clients are not directly connected to server instances handling the data queries and modifications. And, because clients are directly connected to only a single server instance, moving them to another server instances is easier since only one connection needs to be remade. (Dieckmann 2013)

The argument for differentiating login server is not to bring down other services while processing many parallel logins. Login is usually completely separate system from the other game logic which leads it to be easily outsourced to a different server application. Players are also usually only temporarily connected to login service while other services might need constant socket connection—which also supports separating login to its own service. (Dieckmann 2013)

Requirements for a gameplay logic server differ from game to game; there can even be many types of gameplay logic server applications inside one single game. Some parts of the game might be asynchronous, meaning that player does not need immediate response from server to be able to proceed in the game. For instance, in some competitive games, actions of a single player need to be confirmed by the server in order to prevent cheating. While on the other hand, some parts of the game logic might be synchronous (with higher requirements for timeliness). Also, some parts of the game might need more computation on the server-side than others. Aforementioned reasons implicate that gameplay logic should be separated into different server applications depending on resource consumption and real-time requirements—to enable scaling the service on the parts where is needed. Components handling logic with high real-time requirements (along with the proxy server if that is part of the architecture) should be placed close to the players. And the number of server application instances for specific part of the gameplay logic naturally depends on frequency and computational demand of the feature (or set of features).

Obviously, the database should also be separated from other services to ensure scalability of the application and prepare for the almost inevitable software and hardware failures which would otherwise compromise the data. Differentiating the data completely from the logic can make the gameplay logic servers completely stateless. It means that no information of the player is stored on the logic servers, not even temporarily, but it is fetched from the database (or memory cache) each time player operations need to be performed on server-side. The great benefit that follows is that the server applications can be replaced on demand without disrupting the service.

As with login server, also the asset server is usually accessed only by the time player connects to the game. After downloading assets, there is no need for the player to hold connection to the asset server. Downloading assets also creates significant network load for the server distributing the assets. These two reasons combined make a valid argument for separating the asset distribution from other services (Dieckmann 2013). On cloud environment the asset server does not necessarily need to be a computational server application; links can be provided to publicly readable data in cloud storage repository.

Memory cache is implemented to prevent other services from accessing database every time the data needs to be manipulated. Constant updates to database can lead into slowdowns in the game. Games are write-heavy applications and writing to database is generally considerably slower than reading from the database. Implementing a memory cache layer before the database makes writing into the actual database asynchronous process. It can also relieve stress from other servers which are preparing queries to the data. Since the process does not need to be synchronous anymore variety of optimizations can be performed such as collecting queries into batches before executing them.

In addition to these components, there can be some more game specific modules which can be separated from other components—such as a chat server which would be taking care of communications between players. An example of creating a server architecture utilizing some of these aforementioned components is presented in the next section.

5 Last Planets

This section presents the chosen networking middleware and the architectural solution of Last Planets. The practical reasons behind the technology related decisions are reviewed, and the technologies utilized are described more in detail.

5.1 Communication Middleware

The library that was chosen to handle network communication in Last Planets is Photon Server, developed by Exit Games. Photon is widely used networking middleware which is also used by big-name developers such as Warner Bros Games and Codemasters. Photon is designed for games using client-server architecture, and the main reasons for choosing Photon for Last Planets are comprehensive set of features, good documentation and support, ease of use, and compatibility with all major mobile platforms—iOS, Android and Windows Phone.

Since human resources available for the project are limited to a single developer, developing the backend from ground up, for a relatively large scale MMO game, is not a feasible option—at least not within a short timeframe of six months reserved for developing Last Planets ready for global launch. The quality of documentation played an important role when Photon Server was chosen: it is well written and there are multiple example solutions for MMO games. In fact, Exit Games provides fully functional applications, including such features as load-balancing and support for in-game rooms, which can be used as such for many games. In Last Planets, the communication layer—operations, responses, and events—acts as a foundation for the server applications.

Exit Games offer running the Photon Server in their own cloud service, but running the Photon Server on Microsoft Azure was chosen because Exit Games' cloud service does not allow writing custom code on the server-side. Custom code allows creating authoritative server which gives control over cheating. Photon Server can be extended with C# code which makes it naturally well-paired with Unity since some of the code can be shared between client and the server. For Last Planets, all of the game logic that needs validation from server is written only once in a way that it can be run on client and on server-side.

For communication, Photon Server provides reliable UDP, Binary TCP, WebSocket, and HTTP protocols. Reliable is UDP is pertinent for Last Planets for facilitating real-time gameplay scenarios. In combat sequence, both players are sending input data to server which is running the main simulation; losing the inputs due to dropped UDP packets would result in major problems in synchronizing the game play simulation with server and unsatisfactory experience for the player. Using TCP connection is also not a feasible solution due to reasons explained in chapter describing reliable and unreliable messaging.

Photon supports virtual connection over UDP. It allows Last Planets to provide updates about the game-world without players constantly polling for new information. Also, virtual connection lets the game keep track of online status of the players which is useful for chat and players interacting with their alliance members.

Photon offers a possibility to simulate varying internet conditions. For monitoring the Photon framework is using log4net logging framework which is a widely used tool and well supported by other 3rd party tools such as Logstash which is used for collecting, parsing, and storing logs in centralized location. The centralized logging is explained in further detail in next chapter. Photon also comes with the capability to collect windows performance counters and internal counters, all of which can be inspected on a pre-made web interface.

Photon also helps considerably in reducing bandwidth consumption. It uses its own binary protocol for efficient data compression. It also supports receiver groups and priority levels for messages which is especially useful for MMO game where information needs to be sent only to players to whom it is relevant. In Last Planets, players can perform many actions which affect other players' game directly; immediate notifications from server to player are done using receiver groups. Photon also enables developer to choose per message basis which ones should be sent reliably and/or with encryption. Therefore, bandwidth can be saved by using a heavier protocol only when it is needed.

5.2 Server Architecture

This section explains each of the components used in the backend architecture of Last Planets. The following sections review their main responsibilities and main features.

5.2.1 General

All game server applications are running on virtual machines with Windows Server 2012 r2 operating system. Game servers are deployed in cloud services based on their type and geographic location. For example, all the proxy servers in Northern Europe will be in the same cloud service. In Azure, Cloud services provide automatic scaling, and the traffic inside cloud service can be load balanced.

Automatic scaling can be set to trigger when a certain condition is met: either when CPU usage level rises above a threshold, or when number of messages queued for the virtual machine grows too long. The decision of which one to use depends on how powerful virtual machines are being used and what are the requirements of the game server application in terms of CPU and network usage: if the CPU limits are likely to be met faster, the condition should be CPU usage, or vice versa, if game creates a lot of traffic, queued messages should be used as a condition to boot new virtual machines.

Load balancing in Last Planets is done on two levels: on DNS level and on network level. On DNS level, network traffic is directed to different cloud services located in different data centers around the world based on which one the player has the best connection with. DNS level load balancing is done with Azure traffic manager; the traffic manager is configured with “performance” setting which achieves the aforementioned behavior. Azure traffic manager does not actually direct the traffic; the game client makes a query to the traffic manager which returns the IP-address for the most suitable proxy server for the client. By the same token, on network level (inside a cloud service) the load balancing is done with Azure load balancer. Traffic is distributed evenly across virtual machines (round robin) in the cloud service. In addition to load balancing, virtual machines in same cloud service are configured to be in the same availability set. In Azure, when virtual machines are in an availability set, it is ensured that at least some of them are available during planned updates in the service or during hardware failures.

Deploying and managing the infrastructure is done by a configuration management tool called Chef. With Chef, the state of the virtual machines can be written as code (Ruby, to be precise) which means all the required applications, dependencies, and configurations can be controlled remotely and stored in version control. Chef also supports managing different environments: for Last Planets, the servers are either part of development, staging, or production environment; different environment ensure that development can be continued without disrupting the service. Once all of the environments and roles configured, virtual machines can be provisioned to Azure with a single command and be up and running in 10 to 20 minutes. Figure 2 illustrates the components in the server architecture of Last Planets.

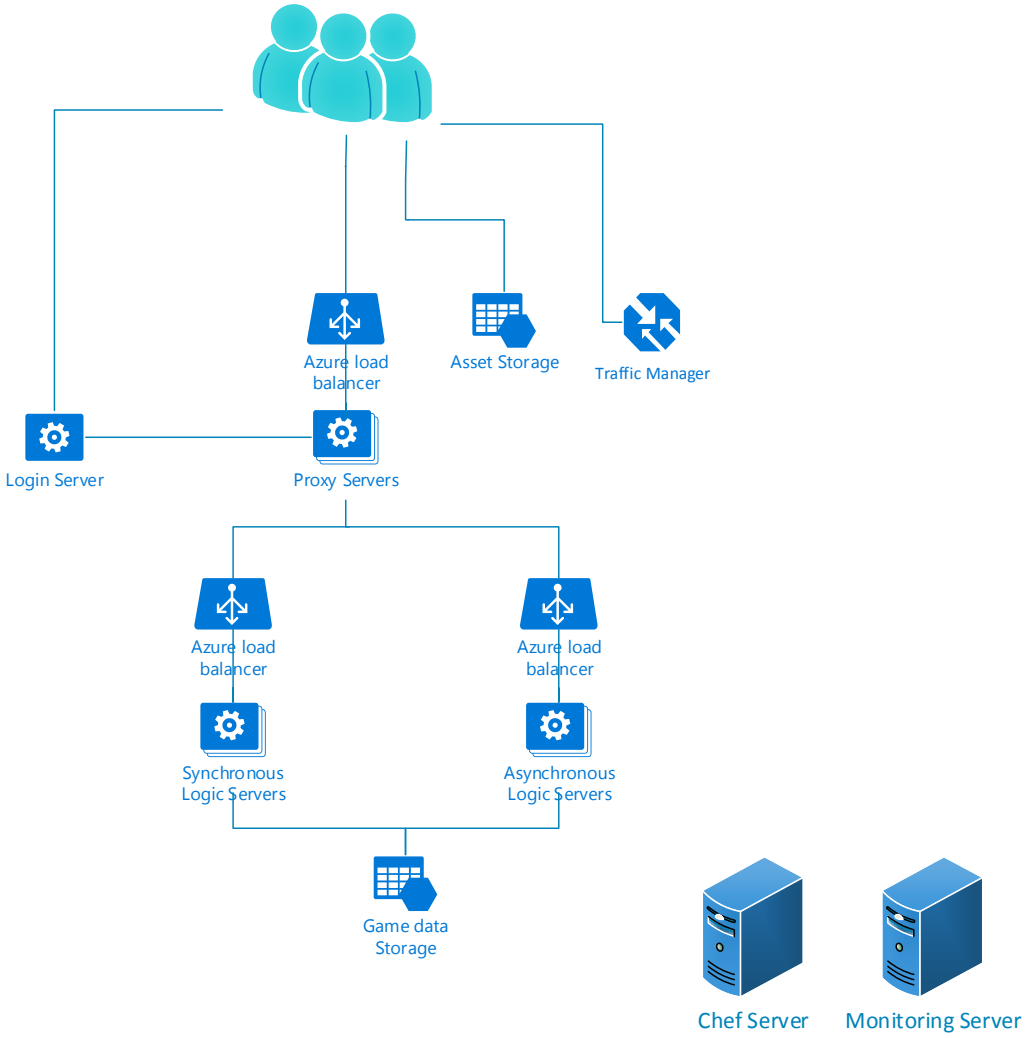

Figure 2. An illustration of the components in Last Planets server architecture. Chef server and monitoring server are connected to all of the computational servers: proxy servers, login server, and game logic servers.

Centralized logging is implemented using ELK stack. ELK stack comprises of Elasticsearch, Logstash, and Kibana. Elasticsearch is a real-time search and analytics engine, Logstash is used to collect and parse the data, and Kibana is Elasticsearch's data visualization engine which allows interacting with the data via custom dashboard. (Sissel)

Each of the virtual machines running the server applications is also running a Logstash-client; it collects windows system events and the application logs and sends them to a central server also running Logstash. The central server parses the data to make it support querying better, and then saves it to elastic search which indexes the data. After processing, the data can be reviewed from Kibana dashboard.

5.2.2 Proxy Server

A proxy server is the server on which players are directly connected with virtual UDP socket connection. The proxy server takes load off from other servers by managing client connections. The proxy server is not completely stateless; when a player performs login action through login server, the login server informs the proxy server which the player will connect and that the proxy server caches the player's information. When the player starts making queries to the server, the proxy server can make a connection between the id of the player's connection and the player's information. Moving players to other server instances is relatively simple: only a single connection needs to be changed and the player's information is fetched to the new server—or in the worst case if the server crashes unexpectedly, the player needs to login again.

Also, the proxy server is part of the architecture for improving security. Since the proxy server is the only one of the game servers that is open to communications from outside and it is not directly connected to the game data storage, the data is not easily compromised.

5.2.3 Login Server

Currently the login server in Last Planets supports email login and Facebook login. A client sends an encrypted login request to the server with both email and password, or the access token provided by the Facebook API. Also, a variable determining whether a new account should be created is passed to the server. After the login has been suc-

cessfully established, the id of the player's connection, known only to the server, is connected with the player's id that can be used to load the player's information from the database.

Since the login service is not dependent on other backend services it naturally leads to separating it to a different server application. The argument for differentiating the login server is not to bring down other services while processing many parallel logins.

5.2.4 Azure Storage

The persistence layer for the system comes from Azure Table Storage; it is a NoSQL data store where gameplay related data is stored. The data structures to be saved in the data store are designed following the best practices for Azure Table Storage and thus storage account is expected to achieve following scalability targets defined by Microsoft:

- Total Account Capacity: 500 TB
- Total Request Rate (assuming 1KB object size): Up to 20,000 entities or messages per second
- Total Bandwidth for a Geo-Redundant Storage Account: Inbound up to 10 gigabits per second and outbound up to 20 gigabits per second

Since the Table Storage is very fast to access, currently all data for Last Planets is in one centralized location. If stress testing the game proves this approach too slow, a memory cache layer is implemented in each of the geographic locations for faster data access.

The solution for distributing updates for static content in Last Planets ended up being Azure blob storage. A typical pattern in cloud development is to have a publicly readable repository in a cloud storage where the assets reside. This pattern reduces the need for potentially expensive compute instances in the cloud.

The game client prompts the server for links for the most recent assets and downloads them from the cloud. A single cloud blob in Azure Storage has scalability target of up to 500 requests per second and can be scaled up by using a content delivery network which caches the contents in multiple datacenters around the world.

The static content distribution can be utilized to update assets such as graphics, sounds and music, configuration files, and even game play scripts which are loaded on runtime to provide updates in the game smoothly–without the need to update the game client which can be considerably lengthy process.

The solution can be improved by implementing a pattern where the client is provided with a token that allows restricted access to the content. In addition to obvious security reasons, the number of requests can be controlled better which can prove useful, especially in cloud environment where the traffic can become costly.

5.2.5 Game Logic Servers

The main responsibility for an asynchronous logic server is to process gameplay logic that does not have high requirements for timeliness (e.g. the validity of solo gameplay actions). It also updates the database when player performs actions which manipulate the game world, such as building new buildings.

An asynchronous Logic server is implemented as a completely stateless server. Being stateless means that after the message has been processed, no information about the player is saved on the server. This makes the server an easy component to plug into the architecture and scale on demand—since it does not matter if the player messages are processed by a different server instances, and thus, no player migration between servers is required. Also, if the server crashes, it can be easily replaced by a healthy instance, and it will be ready to go immediately—player data is not lost since it is in database with fast access from where it can be acquired quickly when needed.

Synchronous logic servers run instances of gameplay sequences which need to be performed in real-time. These include scenarios such as the combat. State of the game is held for duration of the gameplay sequence and the result is stored to database afterwards. Synchronous logic server runs a dedicated thread for the gameplay sequence with all the same logic as on the client side. The server acts as unbiased referee of the sequence, and then it confirms that the result of the sequence is acquired with legal means.

6 Results and Conclusions

At the beginning of the project, the author's knowledge about the subject was limited and it was difficult to predict if the goals could be achieved. But as it turns out, with assistance of products and services in the current market, all of the set goals were met successfully.

A networking middleware that meets all the requirements set by the game was found: Photon is fast enough to implement by a single person with reliable and unreliable messaging, virtual connection, automatic flow control, and it fills all of the network simulation and monitoring needs. It is also well optimized for creating a social game, with many concurrent users, using the client-server architecture.

The backend architecture has most of the fundamental blocks in place for world wide deployment. Virtual machines running the servers can be deployed across different data center locations and the traffic directed optimally. A system for patching the servers without taking the game down for maintenance was created: at the time of patching a mirror of the production environment (called a staging environment) is created which is then updated and afterwards the traffic is directed there and old production environment is taken down to prevent it from causing extra costs.

The server architecture is also maintainable by a single person. All the important events on all of the servers can be monitored from a centralized location and changes to the whole infrastructure can be made by making changes to the configuration code managed by Chef. Changes can be deployed automatically without interacting with servers directly.

The game is also well equipped to prevent cheating. Even if the game client is compromised, it does not enable a hacker to cheat within the game since all of the logic is validated by the server. Also, since both the client and server are programmed with C# language, it is possible to share this logic and write it only once—making the code base clearer to read, easier to maintain, faster to extend, and less prone for errors.

6.1 Discussion

It is encouraging to see that with proper planning and a few months of work even a single developer can achieve substantial results in creating a backend infrastructure for a global social mobile game. Currently, it is surprisingly difficult to find a good overview of all the information needed; it played a big part in the decision to write the thesis on this subject. But, after finding all the necessary information for creating high level guidelines for the project, it was proven that after all the most difficult part is the implementation: an endless list of details and small issues to consider.

Overall the project proved to be a great learning experience. Information was gained from many areas not presented here, such as creating a secure cloud service and creating scalable servers using (new in C# 5.0) asynchronous programming.

The information this thesis presents is not going to get outdated very quickly since it is focused on fundamentals which are not likely change in the near future. The application of these fundamentals is the part that will change rapidly with time.

6.2 Future Improvements

The development of Last Planets still continues, both on the client and server -side, but all of the fundamental building blocks are now placed in the architecture. Naturally, when creating a large scale backend infrastructure within a limited time, there remains a lot of room for improvements.

First, implementing a proprietary networking middleware, even if worthy 3rd party solutions are available, is a good way of mitigating risks—as discussed in chapter 3. Also, the fact that the current solution is tied to Azure (it uses many Azure specific services) eliminates the option to smoothly migrate away from Azure if a need to do so arises. An improved solution would be to create solutions that are independent of any cloud platform.

At the current state, the backend is still yet to be properly load tested. It would be a considerable risk to trust the implementation without exposing the game to heavy load testing. Performing the load tests would also bring invaluable information about the

costs that running servers at full scale will generate. Also, Last Planets needs to be tested on a global scale to clarify if a single centralized location for game related data can be accessed quickly enough. If accessing storage in a single geographic location proves to be too slow, the architecture would be expanded by adding a memory cache in each of the cloud regions.

Finally, a deep analysis of possible security threats and planning on how to prevent them will be imperative before globally launching the game.

7 References

1 Armitage, G. & Claypool, M. & Branch, P. (2006): Networking and online games : understanding and engineering multiplayer Internet games. England: John Wiley & Sons Ltd.

2 Albahari, J. & Albahari, B. (2012): C# in a Nutshell. California: O'Reilly Media, Inc.

3 Aldridge, D. (2011): I Shot You First: Networking the Gameplay of HA-LO:REACH[seminar]. Available: http://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking. Accessed: 30 November 2014.

4 Allman, M. & Paxson, V. & Blanton, E. (2009): TPC Congestion Control[Pdf-document]. Available: http://tools.ietf.org/pdf/rfc5681.pdf. Accessed: 1 November 2014

5 Bernier, Y (2001): Half-Life and Team Fortress Networking: Closing the Loop on Scalable Network Gaming Backend Services. Available: http://www.gamasutra.com/view/feature/131577/halflife_and_team_fortress_.php?page=3. Accessed: 18 January 2015.

6 Bondi, A. (2000) Characteristics of scalability and their impact on performance[PDF]. Available: http://dl.acm.org/citation.cfm?doid=350391.350432&CFID=606071429&CFTOKEN=46217597. Accessed: 7 December 2014.

7 Caldwell N. (2000): Defeating Lag With Cubic Splines[online]. Available: http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/defeating-lag-with-cubic-splines-r914. Accessed 11 November 2014

8 Comer, D.E. (2000): Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (4th Edition). New Jersey: Prentice Hall.

9 Dieckmann, M. (2013): A Journey Into MMO Server Architecture [online]. Available: http://www.mmorpg.com/blogs/FaceOfMankind/052013/25185_A-Journey-Into-MMO-Server-Architecture. Accessed: 17 January 2015.

10 Fiedler, G. UDP VS. TCP[online]. Available: http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/. Accessed: 1 November 2014.

11 Fiedler, G. RELIABILITY AND FLOW CONTROL [online]. Available: http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/. Accessed: 1 November 2014.

12 Fiedler, G. Virtual Connection over UDP [online]. Available: http://gafferongames.com/networking-for-game-programmers/virtual-connection-over-udp/. Accessed: 1 November 2014.

13 Frohnmayer, M & Gift, T. The TRIBES Engine Networking Model[online]. Available: http://gamedevs.org/uploads/tribes-networking-model.pdf. Accessed: 1 November 2014.

14 Gärtner, F. (1999): Fundamentals of fault-tolerant distributed computing in asynchronous environments[online]. Available: http://dl.acm.org/citation.cfm?id=311531.311532&coll=DL&dl=GUIDE&CFID=479206446&CFTOKEN=91589027. Accessed: 14 February 2015.

15 Hidenari, S. & Yoshiaki, H. & Hideki, S. Characteristics of UDP Packet Loss: Effect of TCP Traffic[online]. Available: http://www.isoc.org/INET97/proceedings/F3/F3_1.HTM#Results. Accessed: 1 November 2014.

16 Hiitola, K. (2014): Connecting SignalR, Azure and Unity Game Engine[online]. Available: http://channel9.msdn.com/Events/DevShark/DevShark-Finland-2014/Next-Games-Azure. Accessed 10 December 2014.

17 Hook, B. Introduction to Multiplayer Game Programming[online]. Available: http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/IntroductionToMultiplayerGameProgramming. Accessed: 15 November 2014.

18 Hsiao, T. & Yuan, S. (2005): Practical Middleware for Massively Multiplayer Online Games. IEEE Internet Computing 9(5):47-54.

19 Javvin. (2007): Network Protocols Handbook (4th Edition). California: Javvin Press.

20 Jenkins, K. (2004): Designing Secure, Flexible, and High Performance Game Network Architectures [online]. Available: http://gamasutra.com/view/feature/130587/designing_secure_flexible_and_.php. Accessed: 21 October 2014.

21 Mazdeh, A.S. (2013) Choosing the best networking middleware for your Unity based multiplayer or MMO game[online]. Available: http://www.nooparmy.com/index.php/products/educational-materials/19-choosing-the-best-networking-middleware-for-your-unity-multiplayer-or-mmo-game.html. Accessed: 25 October 2014.

22 Murray, D. & Koziniec, T. & Lee, K. & Dixon, M. (2012) Large MTUs and internet performance[online]. Available: http://www.kevin-lee.co.uk/work/research/murray_HSPR_2012.pdf. Accessed: 8 November 2014

23 Pålsson, J. & Pannek, R. & Landing, N. & Petterson, Mattias. (2011): A Generic Game Server[online]. Available: https://jeena.net/t/GGS.pdf. Accessed 14 February 2015.

24 Sissel, J. An Introduction to the ELK stack[online]. Available: https://www.elastic.co/webinars/introduction-elk-stack. Accessed 14 February 2015.

25 Smed, J. & Hakonen, H. (2006) Algorithms and Networking for Computer Games. England: John Wiley & Sons Ltd.

26 Wyatt, P. (2012) Choosing a game network library[online]. Available: http://www.codeofhonor.com/blog/choosing-a-game-network-lib. Accessed: 25 October 2014.