Peer-to-Peer Network Architecture for Massive Online Gaming

Bongani Shongwe

Supervised by:
Mr. Brynn Andrew, Prof. Clint Van Alten, Dr. José Quenum

A Dissertation submitted to the Faculty of Science, University of the Witwatersrand, in fulfillment of the requirements for the degree of Master of Science.

2014

Abstract

Virtual worlds and massive multiplayer online games are amongst the most popular applications on the Internet. In order to host these applications a reliable architecture is required. It is essential for the architecture to handle high user loads, maintain a complex game state, promptly respond to game interactions, and prevent cheating, amongst other properties. Many of today's Massive Multiplayer Online Games (MMOG) use client-server architectures to provide multiplayer service. Clients (players) send their actions to a server. The latter calculates the game state and publishes the information to the clients. Although the client-server architecture has been widely adopted in the past for MMOG, it suffers from many limitations. First, applications based on a client-server architecture are difficult to support and maintain given the dynamic user base of online games. Such architectures do not easily scale (or handle heavy loads). Also, the server constitutes a single point of failure. We argue that peer-to-peer architectures can provide better support for MMOG. Peer-to-peer architectures can enable the user base to scale to a large number. They also limit disruptions experienced by players due to other nodes failing.

This research designs and implements a peer-to-peer architecture for MMOG. The peer-to-peer architecture aims at reducing message latency over the network and on the application layer. We refine the communication between nodes in the architecture to reduce network latency by using SPDY, a protocol designed to reduce web page load time. For the application layer, an event-driven paradigm was used to process messages. Through user load simulation, we show that our peer-to-peer design is able to process and reliably deliver messages in a timely manner. Furthermore, by distributing the work conducted by a game server, our research shows that a peer-to-peer architecture responds quicker to requests compared to client-server models.

Keywords: Peer-to-peer, latency, online gaming, SPDY, scalability, fault-tolerance

Declaration

I, Bongani Shongwe, hereby declare the contents of this dissertation to be my own work. This report is submitted for the degree of Master of Science at the University of the Witwatersrand, Johannesburg. This work has not been submitted to any other university, or for any other degree.

[illegible]

Bongani Shongwe
July 23, 2014

Acknowledgements

I would sincerely like to thank everyone who has assisted me in completing this research. In particular, I am very grateful to my supervisors Dr. José Quenum and Brynn Andrew for all the advice, motivation, and investment given throughout this research. Through them I have obtained a set of skills and knowledge which I would have likely never achieved. I would also like to thank Prof. Clint Van Alten for his assistance during the final stages of my research. To my parents, Michael and Virginia, I'm grateful for their constant advice and unwavering support in my education. I want to thank all of my proofreaders -in alphabetical order- Sello Ralethe, Sonali Parbhoo and Thato Shebe. I appreciate your assistance and support throughout the years. Furthermore, I would also like to thank the Mathematical Science IT laboratories of the University of the Witwatersrand, specifically, Mr. Shunmunga Pillay, for the use of the Hydra cluster and the other computing machines. Finally, I would like to thank the National Research Foundation (NRF) for the financial assistance provided [1].

[1]Opinions expressed and conclusions derived in this documentation are those of the author and are not necessarily to be attributed to the NRF.

Contents

List of Figures

List of Tables

List of Acronyms

ACID Atomicity, Consistency, Isolation, Durability

AOI Area of Interest

API Application Programming Interface

BDB Berkeley DB

CPU Central Processing Unit

DSL Digital Subscriber Line

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

JVM Java Virtual Machine

MMOG Massive Multiplayer Online Games

MSS maximum segment size

NIO Non-blocking Input/Output

NPN Next Protocol Negotiation

RDBMS Relational Database Management System

RTT Round Trip Time

SBT Simple Build Tool

SLA Service-Level Agreement

SLF4J Simple Logging Facade for Java

SQL Structured Query Language

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UID Unique Identifier

URI Uniform Resource Identifier

VoIP Voice over Internet Protocol

XML Extensible Markup Language

Chapter 1

Introduction

Over the past decade video gaming has become a fast growing industry, making it a multi-billion dollar business [Biscotti *et al.* 2011]. A major attraction in video gaming is the ability to play and interact with other players over the Internet. Today millions of videos gamers make use of online gaming communities, to play against people from different regions of the globe [Neumann *et al.* 2007]. One of most popular online games, World of Warcraft, provides a perfect example of the growth of online gaming communities. In 2006, World of Warcraft had over 6 million subscribers, in 2011, this number grew to 11.4 million subscribers [Ziebart 2011].

Videos games are also no longer just forms of electronic gaming designed for personal computers or gaming consoles. Handheld devices such as cellphones and tablet computers can provide gaming entertainment. The pursuit of entertainment is not the only reason for the rapid growth of video games. Videos games also provide domains for education and social development. In recent years there has been a rise in the number of organized video gaming tournaments, colloquially known as eSports (electronic sports), in which professional video gamers compete. Popular tournaments (such as the World Cyber Games[1]) have corporate sponsorship, media coverage and cash prizes for the competitors. At the 2013 International Dota 2 tournament, the winning team received over $1.4 million [Suszek 2013].

As online gaming communities continue to grow, game developers have to make provision for more resources with the release of new games. Many Massive Multiplayer Online Games (MMOG) make use of a central server or a cluster of servers to provide the multiplayer service. Recently, game developers have also adopted the concept of content on demand through cloud-based game streaming services [Shea *et al.* 2013]. Client-server architectures allow reliable computers to handle tasks such as player access control, game state management, uniform updates, amongst other features.

However, the operation of a game server is costly and has numerous drawbacks. For instance, client-server architectures generally do not scale well. Adequate resources must be provisioned in order to support all the players. If the game servers are overloaded, the users cannot be supported. If the number of resources are above the current demand, the game company faces to lose money due to unused resources. For example, the launch of Diablo III, a role-player game, illustrated the dilemma faced with the provision of servers [Gilbert 2012]. The game was so highly anticipated that, during the week of release, many users were unable to log-in and play the game due to the heavy load the servers experienced.

The greatest drawback in the client-server architecture is that the server represents a single point of failure. If the server fails, users are unable to play games online. A recent and well documented event relating to game server failure is that of the PlayStation Network outage in 2011 [Richmond and Williams 2011]. The PlayStation Network is an online multiplayer game and digital media delivery service for the PlayStation 3, PlayStation 4, PlayStation Portable, and PlayStation Vita gaming consoles. In April 2011, the PlayStation Network was “compromised” through external intrusion. This led to the PlayStation Network being shut down for a month, leaving over 77 million PlayStation Network users unable to play online multiplayer games or access other PlayStation Network services. Given the growth of online gaming, scalability and reliability pose grim challenges awaiting the gaming industry.

[1]www.wcg.com

With increasing home bandwidth and computational power, decentralized peer-to-peer architectures are becoming a viable option to solve the issues which can be experienced in client-server gaming architectures. Peer-to-peer applications seek to use easily accessible commodity computers to provide a service. Many of today's traffic intensive applications are based on peer-to-peer architectures. These applications include content distribution (for example Gnutella, LimeWire, BitTorrent), Internet telephony (Skype), and streaming of television content (PPLive). Peer-to-peer networks have the key characteristic of being able to scale to a large number of nodes, distributing the computational and network load between the peers. In a client-server architecture, the performance of an application will decline as the number of users grow. Due to limited resources, such as hardware and bandwidth, a server can maintain set number of user requests before the quality of service deteriorates. Peer-to-peer applications, such as Chord (a distributed hash table) [Stoica *et al.* 2001], are able to maintain (or even improve) the performance of a service as more users participate in the network. By distributing an applications services, each peer contributes resources to maintain an applications quality of service. Hosts in peer-to-peer architectures can also be self-organising [Knutsson *et al.* 2004]. As peers join and leave the network, peers can organise themselves into ad-hoc groups to communicate, collaborate and share hardware resources to complete a task [Ding *et al.* 2003; Rowstron and Druschel 2001; Stoica *et al.* 2001]. This dynamic re-organisation of peers is often transparent to the users.

Peer-to-peer applications are cost efficient because the application utilizes existing user resources. For example, file sharing and data storage services require high-end machines to serve the users [Sadalage and Fowler 2013]. Peer-to-peer networks use commodity machines to host file sharing and data storage services [Ding *et al.* 2003; Sadalage and Fowler 2013]. There is also no need for specialist staff such as network administrators to maintain a peer-to-peer application. Users usually monitor and administrate the usage of a peer-to-peer application, or a self-administrating protocol can be built into the peer-to-peer application. BitTorrent, a peer-to-peer file sharing service, contains a self-administrating *tit for tat* protocol [Cohen 2003]. Users in the BitTorrent system are required to share files which they have obtained from downloading from another user, this ensures that a copy of the file is always available. The *tit for tat* protocol encourages fair file trading amongst the users by allowing a user to continue downloading files from other peers as long as the user also contributes to sharing of files. Peer-to-peer applications also aim to solve fault-tolerance. If a peer were to fail or exit the network, the application should continue to operate. For example, BitTorrent systems are tolerant against peers leaving the network [Cohen 2003]. If a peer containing a specific file were to leave the system, the remaining peers contain a replication of that file that can be retrieved by any requesting peer.

Based on the hypothesis that centralized online gaming architectures do not scale well, are costly to maintain and present a single point of failure, peer-to-peer architectures provide intriguing properties to solve client-server based dilemmas. Despite the cost of affordable scalability and fault-tolerance offered by peer-to-peer architectures, there are certain drawbacks that hinder certain applications being implemented in a peer-to-peer environment. With respect to gaming, these include complex data management, preventing cheating, obtaining low latency and difficulty in adapting existing game applications into a peer-to-peer environment. Also, different games have different requirements that should be met by the underlying network architecture to ensure that the game can be played interactively online by the users [Fiedler 2008; Yahyavi and Kemme 2013]. This makes it difficult to design a general purpose peer-to-peer gaming architecture. Given the above challenges in utilizing peer-to-peer architectures for online gaming, a growing interest has arisen in peer-to-peer architectures for MMOG and virtual worlds [Krause 2008; Neumann *et al.* 2007; Schiele *et al.* 2007; Yahyavi and Kemme 2013]. However, there is no consensus on which architecture design will serve a peer-to-peer online network best. Also, many of the challenges for peer-to-peer online gaming have not been fully addressed.

1.1 Research Scope

This dissertation aims to extend the on-going research in peer-to-peer gaming networks. MMOG encounter frequent updates that must be propagated to the users under a set time frame to present a highly interactive game world. If the online gaming network fails to process and disseminate messages under the set time frame, players experience a decrease in game play interactivity that often halts the game [Howland 1999]. This research focuses on minimizing the delay that can occur within peer-to-peer gaming networks in order to present a smooth and natural interaction experience in an online game. Specifically, we aim to reduce the time taken to process network messages and reliably propagate messages to the players over the network. In context of network communication protocols, the standard protocols either do not provide guarantee of message delivery or are likely to delay message transportation. The SPDY protocol is used to refine communication on the delay prone Transmission Control Protocol, allowing for messages to be delivered in a timely manner whilst providing message delivery guarantee.

In context of processing network messages, a highly concurrent paradigm is required to reduce the time taken to process messages. We adopt the actor event-driven paradigm to complete tasks asynchronously by passing messages between processes. Akka, an actor-based framework, is used to rapidly process the network messages. Also, by implementing a peer-to-peer architecture, we face a challenge of managing the game state in a distributed environment and ensuring fault-tolerance against nodes abruptly leaving the network. The concepts of non-relational distributed systems are adopted to provide a highly available and distributed data management system. From this, we employ a distributed key-value system, Project Voldemort, to manage the game state data and automatically replicate data across several nodes for fault-tolerance.

The main contribution of this research is a generic peer-to-peer network application aimed at hosting MMOG. The peer-to-peer network application could also be used for other applications that have similar requirements as online games. The peer-to-peer gaming network presented in this dissertation contains three main functions: a distributed storage system to manage the game state and game date, a publish and subscribe service used to propagate messages to the users, and a service routing function used to direct messages to the game application for further processing. The network application uses SPDY to reliably transport messages over the network. The Akka actor toolkit is used to rapidly process the network messages and Project Voldemort is used to manage the game state data. By distributing a game servers functions (such as message dissemination and game state management) into a peer-to-peer architecture, this research shows that a distributed system can be used to decrease the message response time on a system experiencing high user loads. The SPDY protocol, along with the Akka based network application is evaluated. The evaluation shows that SPDY is able to reduce the delay experienced on the Transmission Control Protocol by simultaneously retrieving requests and that Akka is able to reduce the time taken to process a message simply by increasing the number of instances of an actor.

1.2 Dissertation Overview

This dissertation consists of 7 chapters in addition to the appendices and references. The dissertation is structured as follows:

- **Chapter 2** discusses the background related to the research. The background consists of two segments. The Chapter provides an in-depth discussion of the design principles used in Massive Multiplayer Online Games. In particular, the chapter presents design principles created to the reduce latency in online games to provide high interactivity amongst the players and the game. This chapter also presents the challenges facing peer-to-peer gaming architectures and the message dissemination protocols designed for peer-to-peer games.

- **Chapter 3** provides the specific aims of this research and a series of research questions on which this work is focused on. An overview of the peer-to-peer gaming architecture is presented. Also, the methodology used to answer the research questions is discussed.

- **Chapter 4** discusses the building blocks of our peer-to-peer gaming solution. The chapter reviews challenges and solutions from various computing applications that can be transferred onto peer-to-peer gaming networks. In particular, we review distributed database systems, network delay management of web applications and programming paradigms that support massive concurrency demands. The methodologies used to manage distributed data, reduce network delay, and concurrently process tasks form the basis of the peer-to-peer gaming solution discussed in Chapter 3.

- **Chapter 5** provides a thorough discussion on the design of our peer-to-peer gaming network solution. It also includes a section on how to run the peer-to-peer network.

- **Chapter 6** presents the evaluation results performed on our peer-to-peer gaming solution. The evaluation focuses on the time taken to receive a response from a request message. We examine the following three aspects of our peer-to-peer gaming solution: the time taken to transport a message over the network, the time taken for a single host to process requests and finally the time taken to process requests in a distributed environment.

- **Chapter 7** provides a summary of the research findings and a conclusion to the research document. We also discuss some future research topics.

Chapter 2

Background

2.1 Introduction

In this chapter, we present the design principles of online gaming that form the basis of this research. Section 2.2 presents the evolution of Massive Multiplayer Online Games (MMOG). We focus on the design principles that were developed to reduce game disturbance caused by latency. We also discuss the effects that transport layer protocols have on the interactivity of an online game. In Section 2.3 we investigate the challenges facing peer-to-peer gaming networks. The discussion allows us to highlight issues that we are required to address in this research. In Section 2.4 we extended the research into peer-to-peer systems and discuss current peer-to-peer gaming protocols. We also provide a comparison on the performance of each protocol. We conclude the chapter by briefly discusses hybrid gaming architectures that mix client-server and peer-to-peer architectures.

2.2 Online Gaming

2.2.1 Type of Games

There are various categories which may be used to classify online games. As far as networking is concerned there are four primary game categories: *turn-based games, real-time strategy games, role-playing games, and action games* [Fiedler 2008]. Each category (and individual game) has a set of requirements which a network must meet in order for the game to be played over the network. These requirements often concern the bandwidth and latency network properties [Ng 1997]. The network bandwidth indicates how much data can be transferred at any given time. This gives an indication of how many players can be simultaneously supported. Network latency indicates the time it takes for a data packet to travel across the network from the sender to the receiver. If packets do not arrive in time, the game action will halt, resulting in a visual delay, also known as lag [Howland 1999].

Turn-Based Games

Turn-based games are games in which each player takes a turn playing. Traditionally, when a player is making a move, no other player is allowed to interact with the game. Examples of turn-based games are Chess and Checkers.

Real-Time Strategy Games

Real-time strategy games are centered on the user utilizing resources to develop units (such as an army) in order to defeat other opponents. The term real-time is used because players must attempt to build their resources, defend their bases and launch attacks while knowing that the opponent is doing the same thing; whereas in turn-based strategy games, each player has the time to carefully consider the next move without having to worry about the actions of their opponent. Some popular real-time strategy games are the Starcraft series [Blizzard 2014], the Command and Conquer series [Bell 1998; Honeywell 2000] and the Age of Empires series [Kent 1998; Microsoft 2014a].

Role-Playing Games

Role-playing games (sometimes referred to as RPG) are video games where the player controls the avatar(s) of the protagonist (or adventuring party members) in a fictional world. The role-player video game genre is said to be inspired from tabletop role-playing games such as Dungeons & Dragons [Barton 2007]. Similar to the tabletop counterpart, role-playing video games contain a complex story line, character development, travelling to different places and communicating with other game avatars. Some popular role-playing video games are Diablo [Blizzard 2014], World of Warcraft [Blizzard 2014] and Final Fantasy [Enix 2014].

Action Games

Action-based video games are games that require rapid processing of sensory information and prompt actions. This forces players to make decisions and execute a response in a short time span [Dye *et al.* 2009]. The action game genre itself is very diverse and can be subdivided into several sub-genres such as: first-person shooter games (e.g. Unreal [Epic 2014], Quake [id 2014], Halo [Microsoft 2014b]), fighter games (e.g. Mortal Kombat [Midway 2014], Street Fighter [Capcom 2014], Tekken [Bandai 2014]) and racing games (e.g. Need For Speed [Arts 2014b], Burnout [Arts 2014a]).

2.2.2 Online Gaming Background

Online games were initially deployed on peer-to-peer architectures [Fiedler 2008]. The synchronisation of player's actions was achieved by separating the game into a series of turns and then ensuring the turns happened in lockstep across the peers. On each turn, every machine sent its user's actions to the other machines. Once a machine had received all the messages, it calculated the game state and sent the new user actions to every other machine. While this protocol proved successful for real-time strategy games such as Age of Empires, it failed for action-based games such as Doom [Fiedler 2008]. The lockstep mechanism was the main reason for this failure. In action-based games, the game state changes in short periods of time. Using the lockstep mechanism resulted in action-based games experiencing an extensive amount of lag. Since a machine had to wait to receive messages from every other player, the game action would "halt" until all the messages were received.

The problem underlying the lockstep synchronisation was resolved by using a client-server architecture. Instead of exchanging messages with each other, the players would exchange messages with a dedicated game server (often owned by the game developers). The server simulates the game in distinct time steps referred to as ticks [Valve 2012]. During each tick, the server processes the users input, runs a simulation step on the input according to the game rules, and updates the state of all objects affected by the users' actions. After the tick simulation, the server sends the new game state to the client, who then renders and displays the new game state. Figure 2.1 shows the general processes between a client and server in a networked game. If a player's machine was unable to send or receive messages in time for the next tick, the player would be the only one to experience lag, not every other player. The client-server online gaming architecture grew in popularity and was quickly adopted across a range of games. In the client-server architecture, the server holds all the data of the games mutable objects and maintains a consistent view of the game state.

Server tick rates differ from game to game, mostly dependent on the amount of action that occurs during game play. A higher tick rate increases the simulation precision of the game server, but it also requires more processing power and bandwidth between the clients and the server as more messages are generated and processed. Some common tick rates from first person shooter games are 33, 66 and 100 ticks simulated per second. Clients sample input from devices at the same tick rate as that of the server. The game data sent by clients and servers is often compressed using delta compression in order

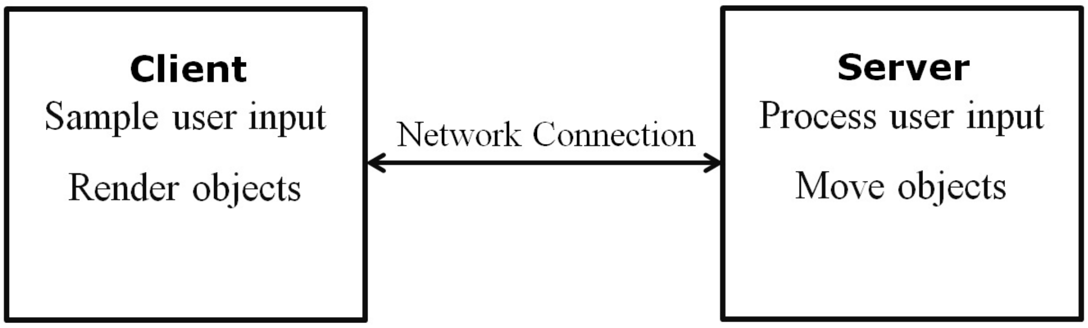


Figure 2.1: General client-server game architecture, adapted from Bernier [2001].

to reduce network load [Mulholland and Hakala 2004]. That is, the host does not send a full snapshot of the gaming world on each tick, but rather only changes of state. For example, if a player were to press a button and not release it, there is no need to send a message to the server on every single tick that the button is still pushed down. The client could rather send a message when the user releases the button. Similarly in the server case, if an avatar were not to move, the server need not notify all the clients about the avatars location on each tick. It would be more sensible to send the avatars coordinates when it moves to another location.

Due to the increase in popularity of online gaming, large game developing companies began incorporating clusters of servers (multi-servers) to host games. Multi-server architectures can be used in two ways [Yahyavi and Kemme 2013]. First, each server runs its own independent instance of the game. Each server is responsible for its own clients and generally the servers do not interact with each other. Several servers of the multi-server are located in different regions around the world. Players are usually assigned to play on the server nearby their geographical location to avoid packets travelling long distances. In the second model, only a single instance of the game exists. The regions (or levels) of the game world are divided, with each region being managed by a single server. Players are in the same game world but only interact with players in the same region. Figure 2.2 gives a visual presentation of this multi-server model. When a player proceeds to another region of the game world a hand-off process occurs between the two servers, moving a player from one server to another. The hand-off process is not always transparent. It often requires the player's avatar to go through a special doorway, portal, or gateway that proceeds to load the player into the next region.

Using clusters of servers is costly. Mulligan *et al.* [2003] notes that acquiring servers to support at least 30,000 users simultaneously can cost upwards of $ 800,000. Furthermore, a single server can only support a defined number of users. If a server becomes flooded with users, the game developers have to make provisions for another server. With region-based multi-servers, bandwidth cost increase further due to the inter-server communication when hand-offs occur.

With the increase of processing power in personal computers and game consoles, and the increase in bandwidth available to homes, different ways of hosting games were envisioned. Instead of having the game server hosting all the games being played, some games allow users to host a game on their machine (acting as the server) [Agarwal and Lorch 2009]. This took some load off the dedicated game server. Algorithms, known as *matchmaking*, were developed in order to select a player's machine that qualified best in hosting a game. Criteria often used in selecting which player's machine to use to host the game range from hardware specification, bandwidth availability and the reputation of the player. If the host machine were to be (intentionally or unintentionally) disconnected, the game activity on all the clients would end. Some matchmaking algorithms simply halt the game while a new host is being selected. The newly selected host gathers game information from all the clients in an attempt to rebuild the game state before resuming the game. This process can take upwards of 15 seconds. *Matchmaking* is also used to allocate players to a server in terms of the round-trip time. A server that is measured to have a small round-trip time communicating with the client is less likely to exhibit issues of latency. Many *matchmaking* systems link players of the same skill level into the same game. This allows players to

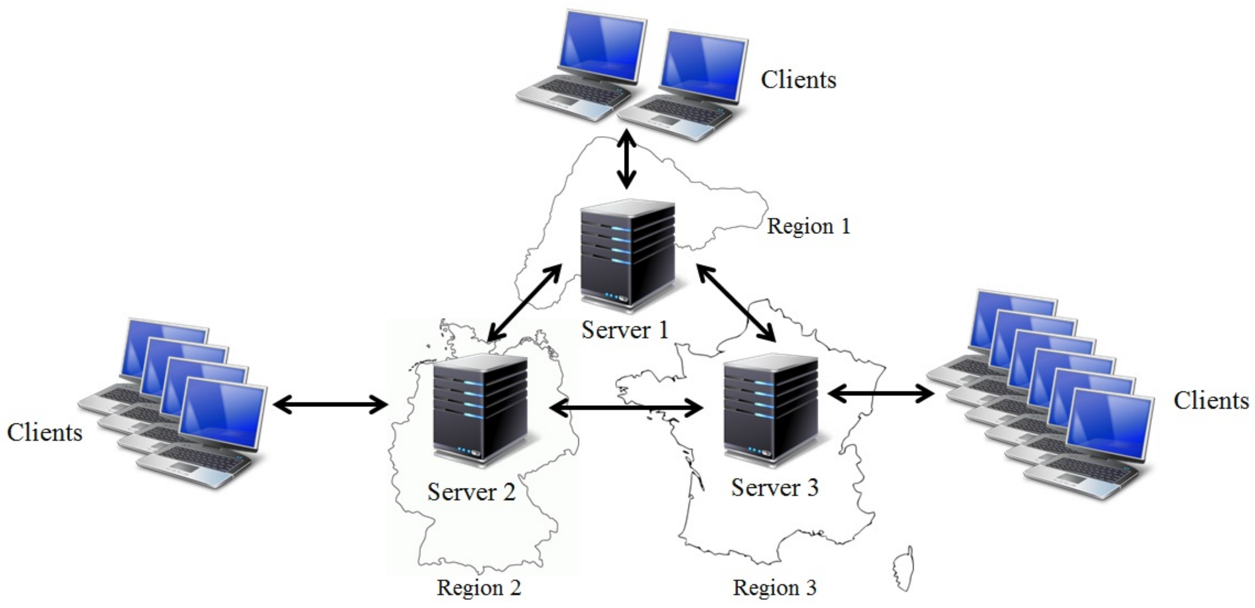


Figure 2.2: Region-based multi-server architecture, adapted from Yahyavi and Kemme [2013] illustration. Each server manages a specific region in the game world.

enjoy a game with players of the same level, rather than being constantly beaten by players on a higher skill level.
Game responsiveness, the latency between user input and its visible feedback in the game world, can be influenced by a number of properties. These include server/client CPU load, simulation tick rate, message data size, network bandwidth, and network packet travelling time. The latency, or the ping rate, is the time taken between the client sending a user command, the server responding to it, and the client receiving the server's response. The lower the latency, the smaller the chances of experiencing lag. In the following sections, we discuss some of the common techniques used in gaming networks to decrease the latency.

Lag Compensation (Dead Reckoning)

During the early stages of client-server gaming systems, lag was often noted in action-based games [Fiedler 2008]. This was caused by the quick transition rate from one game state to another. Latency still proved to be an issue in many multiplayer online games. The time it took for a player to send actions to a server, the server to calculate the next game state and then send the state back to the client would often fall behind the time taken for the player to move on to the next game state.

This issue was resolved by allowing the client to predict the next possible game state [Fiedler 2008]. The client side of the game runs a subset of the server code which is able to predict the position and state of objects in the next game cycle. For example, if a player is observing an object travelling down a hill, by using the game physics engine[1], the client can predict the position of the object from previous states. This technique is referred to as *dead reckoning* [Murphy 2011]. There are instances where the server will not agree with the client's game state prediction. In such cases, the server always has the final decision. A "rewind" technique is then implemented on the client.

Figure 2.3 shows a screenshot from a first person shooter game, Counter-Strike [Valve 2012]. The red hitbox shows the target position on the client where it was before the hit was confirmed. Since then, the target continued to move to the left while the user command about the shooting action was travelling to the server. After the user command arrived, the server restored the target position (blue hitbox) based

[1]The game physics engine is usually implemented on the server side. It can also be available on the client side in order to enable client-side prediction.

on the estimated command execution time. The server traces the shot and confirms the hit (the client sees blood effects). The client and server hitboxes do not match-up exactly because of small precision errors in time measurement. Even a small difference of a few milliseconds can cause an error of several inches for fast-moving objects, though the user never notices due to the fast pace of the game [Valve 2012]. For further explanation on how client prediction and lag compensation work, also how lag compensation is unnoticeable to the player, we refer the interested reader to Valve [2012] and Bernier [2001]. Aldridge [2011] discusses not only lag compensation methods used in Halo Reach, but other techniques such as message prioritization used to reduce latency.

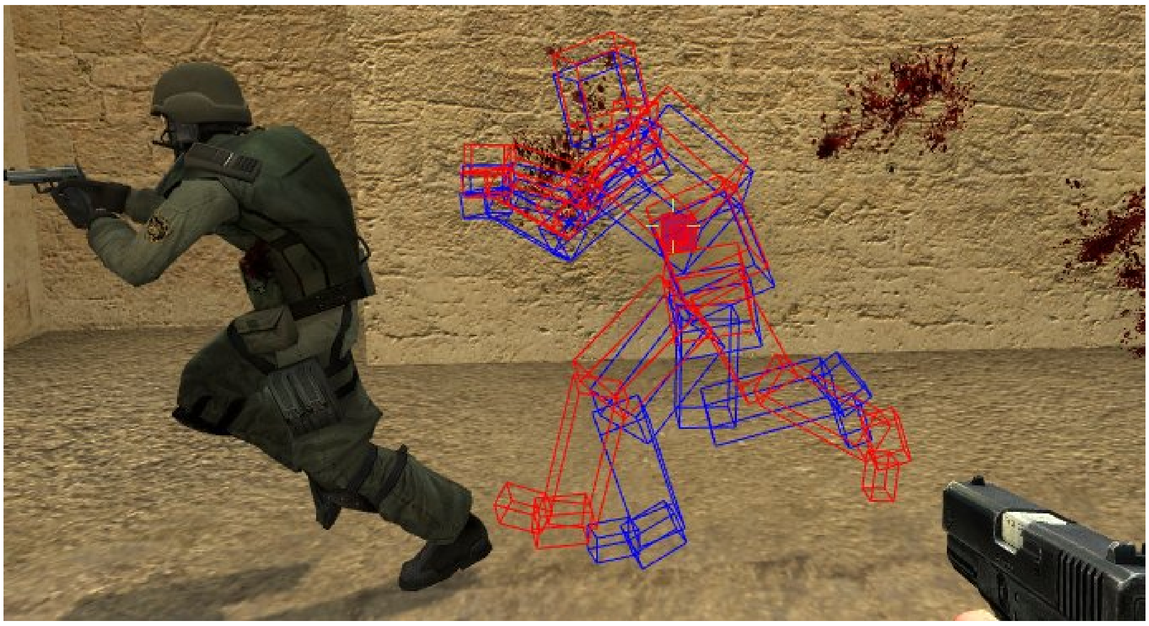


Figure 2.3: Historic client hitboxes (red) versus rewound server hitboxes (blue) [Valve 2012].

Area of Interest (AOI)

As discussed earlier, delta compression was used to decrease the size of messages exchanged between the clients and the server. Another factor which affects latency is the number of messages exchanged between the clients and server. Network congestion can be caused by a large number of messages being exchanged between hosts, resulting in messages being dropped by the network routers. In certain games, a player's avatar only interacts with a small set of the gaming world. The Area of Interest (AOI) specifies the scope of the gaming world that a player should receive information on [Boulanger *et al.* 2006]. This decreases the amount of data a server sends to clients.

2.2.3 Transporting Data within Online Games

The transport layer provides end-to-end communication services for applications [Braden 1989]. There are two primary transport layer protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is a reliable, connection-oriented transport service that provides end-to-end reliable data transfer, same order delivery and flow control [Braden 1989]. UDP is a connectionless transport service [Braden 1989]. Unlike TCP, UDP does not guarantee that packets will arrive in the same order they were sent. There is no flow control and most importantly it does not provide a reliable data transfer service. If an application requires reliable data transfer using UDP, the reliability property is often handled at the application layer.

The transport layer used in an online game is dependent on the type of game. Turn-based games use TCP, also so do some real-time strategy games. For role-player games it is less clear which transport layer protocol should be used as its dependent on the amount of action which may go on during game play

[Griwodz and Halvorsen 2006]. Action-based games such as first person shooters and racing games use UDP [Fiedler 2008]. UDP is preferred over TCP in action-based games due to the cost of TCP's reliable service guarantee. When a TCP packet is discarded by the network, the sending host has to retransmit the packet. But first the sending host has to detect that the packet was dropped. If an acknowledgement for the packet is not received under a set time frame, the protocol on the sending host deduces that packet was discarded and retransmits the packet. During this period, the receiving host may receive newer packets sent before the packet retransmission. Due to TCP's property of message order, segments are held in the buffer until the original data stream can be reconstructed. The delay means a game is unable to receive new messages until the missing packet is retransmitted and received. In action-based games where state is rapidly changing, the delay in waiting for an "old" packet to be retransmitted can cause a huge amount of lag. UDP does not present this issue because packets are not buffered in the transport layer and are immediately sent to the application layer when received.

Action-based games tolerate a certain amount of packet loss when using the UDP transport layer. Action-based games are interested in receiving the latest information. As long as new packets of information are received, lag compensation methods may be used to predict past actions. Message delivery guarantee and flow control are still desirable properties with games that use UDP. If an important event (such as someone winning the game) was contained in a packet that was discarded, some form of reliability would be required to ensure that the packet is delivered to a host. Flow control would also be required in order to prevent hosts from burdening network nodes such as routers with messages, further causing messages to be discarded. Games which use UDP need to implement their own form of message delivery guarantee and flow control in the application layer. There are several UDP wrappers, such as Enet[2], HawkNL[3], PLib/Medusa[4], SDL_net[5], that offer message delivery guarantee, flow control and other features specifically for online games.

2.3 Challenges in Peer-To-Peer Gaming Networks

Though peer-to-peer gaming architectures seem promising in solving scalability issues in client-server networks, they also produce several requirements which must be addressed in order to provide a reputable service. Challenges facing peer-to-peer gaming networks include distributed storage and game state consistency, optimization of message delays, fault-tolerance management and protection against cheating [Neumann *et al.* 2007]. These issues are not limited to peer-to-peer gaming architectures, but are shared by various distributed computing applications such as file sharing services and distributed storage systems. We focus on the requirements of game distributed systems in terms of game state management, scalability, fault-tolerance, delay management and cheating. In the following section, we discuss each property in detail.

2.3.1 Game State Management

A game is characterized by a set of states [Neumann *et al.* 2007]. These states are modified as the game progresses, for example a player's actions will cause the game state to change. With multiplayer games, it is important that the game state is consistent amongst all the players. That is, all the players must reflect the same game world view. In a client-server architecture, it is easy to manage the game state. The server is able to keep the game state consistent by being the only device interacting and modifying the game state. The game state which players view is that state which was calculated by the server. In a peer-to-peer architecture, it becomes difficult to manage the game state. More than one node could interact with the game state, which could cause the game state becoming inconsistent. Section 2.4 discusses peer-to-peer approaches for exchanging game state information.

[2]enet.bespin.org
[3]hawksoft.com/hawknl
[4]plib.sourceforge.net
[5]www.libsdl.org

2.3.2 Scalability

Scalability is the ability of a computer system to continue to function well when it grows in size or volume [Ding *et al.* 2003]. For a network application, peer-to-peer networks are enticing due to the scaling property. Bandwidth, processing power and storage can be distributed among nodes allowing more participants to utilize the application. Peer-to-peer applications can experience excessive inter-node communication due to scaling, introducing excessive bandwidth consumption and latency. For example, if a peer wished to retrieve data from the peer-to-peer network, but without knowledge of where the data could be obtained, a query message could be sent out to all the peers. Given a large group of peers, this option could easily clutter the network. Gnutella, a file sharing service, was one of the most well-known applications to use this protocol [Ding *et al.* 2003]. Peer-to-peer protocols alleviate this issue by reducing the scope of the query and exchanging messages with relevant neighbouring peers. Freenet adopted the Gnutella system but used a key-based protocol (similar to a distributed hash table) to query nodes. The key-based protocol reduces the number of queries, therefore reducing the bandwidth congestion.

2.3.3 Fault-Tolerance

Fault-tolerance is the property that allows a computer system to continue to operate correctly in the event of one or more of the systems components fail. As more nodes join a peer-to-peer network, the possibility of a node abruptly failing and leaving the network increases. Peer-to-peer applications must be resilient to a peer's failure in order to ensure service availability and provide smooth user experience. Napster, one of the most well-known file sharing services exhibited the importance of fault tolerance [Ding *et al.* 2003]. Napster allowed users to directly exchange files with other users. A central server indexed the files which users shared. When the central server malfunctioned, the whole Napster application was inoperable. Peer-to-peer gaming protocols often make use of replication as a form of fault-tolerance [Hampel *et al.* 2006; Yahyavi and Kemme 2013]. A peer that maintains a portion of the distributed state data will replicate the data to another peer. If the peer maintaining the data happens to fail, the data can still be accessible from the peer containing a copy of the data. Various forms of replication techniques exist which are dependent on the protocol used in the peer-to-peer gaming architecture. It is also important to note that each player in the game contains a local copy of game state data pertaining to the players AOI.

2.3.4 Delay Management

As discussed in detail in Section 2.2, delay in game responsiveness is often caused by latency. The delay could be caused by the time taken to transport and process information over the network, *network delay*, or the time taken to account for all the players actions and synchronise the state, *state synchronisation* [Neumann *et al.* 2007]. A peer-to-peer network could add on to the network delay. With the client-server model, a message takes a single hop (from client to server or vice versa) for the message to reach its destination. In a peer-to-peer network, dependent on the network topology and messaging protocol, a message may take several node hops before reaching its destination. A peer-to-peer gaming architecture must seek to minimize this delay.

2.3.5 Cheating

In this research, we use the Neumann *et al.* [2007] definition of cheating: "an unauthorized interaction with the game system aimed at offering an advantage to the cheater". Cheating in general is a huge problem in any game architecture. Players who cheat often gain an unfair advantage that could result in frustrated legitimate players cheating as well or leaving the game. When players are found to be cheating by the game developers, it often results in the player being banned from playing the game or resetting the players progress. In 2010, over 5,000 players who were found to be cheating were banned from playing Starcraft 2 [Entertainment 2010]. Disciplinary action, alongside game patches are some of the techniques game developers employ to deal with players who cheat. There are several methods that can be used to cheat, below we give three which are prone to peer-to-peer gaming networks [Neumann *et al.* 2007; Yahyavi and Kemme 2013]:

- Accessing unauthorized information: A player can intercept and examine information (for example other players' position) that is not supposed to be disclosed to the player. The player can use this information to devise strategies in order to guarantee a win. Peer-to-peer architectures exhibit this issue due to exchanging game state information with peers, whereas in server based architectures the server communicates directly with the targeted client.
- Interrupting information dissemination: A cheater may target the game state of other players by delaying, dropping, corrupting, changing the rate of updates or broadcasting inaccurate information. This results in other players computing an incorrect game state, likely decreasing the cheaters chance of being targeted. Client-server architectures benefit from a single server managing the game state, ensuring game state integrity. Peer-to-peer gaming architectures distribute the game state amongst the peers, allowing a cheater to manipulate the game state of other players.
- Defying game rules: A cheater may target the game application by circumventing the game rules. This can be done by reverse-engineering the game to obtain information on how it works and modifying the game code. Cheating that involves hacking the game application or network system can lead to instability and performance issues with the game, furthermore it raises security issues. A player could also use third party software to gain an advantage. For example, an aimbot which automatically aims a players weapon in the correct direction of an enemy's avatar [Yu *et al.* 2012].

Cheat detection and prevention are common techniques used by game developers to battle cheating. Cheat prevention methods eliminate (or at least reduce) the possibility of cheating. Cheat detection methods actively seek and punish cheaters, deterring other players from cheating. In server-based gaming architectures, cheat detection and prevention is relatively simple as a trustworthy server can perform the operations. Peer-to-peer gaming architectures present issues such as finding methods to apply cheat resistant operations on a large number of peers. Another issue is determining where the cheat resistant mechanisms should be placed in the architecture. If a cheater has access to the cheat prevention and detection mechanisms, the cheater could devise a method of being undetected. Reputation systems are a form of calculating which peer is most trustworthy, and it also prevents false-positive cheat identification resulting in a player being punished by mistake [Yahyavi and Kemme 2013].

2.4 Overview of Current Peer-To-Peer Gaming Protocols

In Section 2.3, challenges facing peer-to-peer gaming networks were addressed. In this section, we discuss peer-to-peer gaming protocols for message and state dissemination. The protocols are classified into the following three categories: Application Layer Multicast, Supernode Control and Mutual Notification.

2.4.1 Application Layer Multicast

The Application Layer Multicast protocol segments the game world into regions, with each region containing a dedicated multicast group. If an event occurs in a certain region, messages are sent to players who have subscribed to the region. Messages are delivered to players using a multicast tree. The peer-to-peer gaming system discussed in Knutsson *et al.* [2004] is an example of a peer-to-peer gaming infrastructures that uses the Application Layer Multicast protocol. Iimura *et al.* [2004] states that the Application Layer Multicast protocol may incur unnecessary network delay due to the number of hops messages take when broadcasting through the multicast tree. Therefore, the application layer multicast protocol is unsuitable for latency sensitive games, especially when there are a large number of participants.

2.4.2 Supernode Control

As in the Application Layer Multicast protocol, the game world is segmented into regions. A node is then selected to co-ordinate events within the region and becomes a supernode. Players who are in a certain region send their status to a supernode to which they have registered. The supernode upon receiving the status calculates the next game state and publishes the game state to the registered players. Although this model shares some similarities with the client-server model, it differs in the scale of operations in that multiple supernodes attend to the players as opposed to just a single node (the server). Figure 2.4 illustrates the Supernode Control protocol. The systems discussed in Bharambe *et al.* [2006] and Yamamoto *et al.* [2005] are examples of gaming peer-to-peer infrastructures which make use of the Supernode Control protocol. Supernodes may be overwhelmed with servicing crowded regions. If too many players are inside a region the supernode will struggle to receive and deliver messages in time. In such cases, the region will be divided further into smaller regions and new supernodes will be elected to assist in balancing the load.

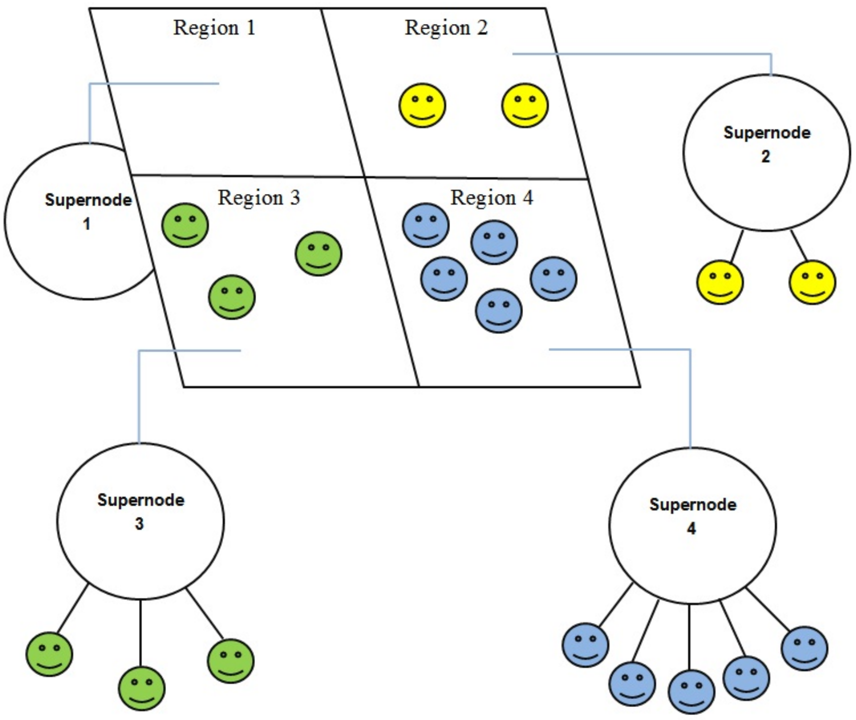


Figure 2.4: Illustration of the Supernode Control protocol.

2.4.3 Mutual Notification

In contrast to the Supernode Control and Application Layer Multicast protocols, the Mutual Notification protocol does not segment the game world into regions. Instead, players send messages directly to other players who are in their area of interest. The peer-to-peer gaming systems discussed in [Keller and Simon 2002] and Hu *et al.* [2006] are examples of systems that use Mutual Notification. Figure 2.5 illustrates the mutual notification protocol. The player in the center of the circle sends messages directly to its nearest neighbours (the solid circles). Messages about the player's actions are 'gossiped' to the other players outside the circle (the hollow circles). Compared to the Application Layer Multicast and Supernode Control protocols, the Mutual Notification protocol minimizes the delay of messages by communicating directly with other players. Essentially, there is only one hop for a message to reach its destination, unlike the other protocols where there will be at least two hops. However, the Mutual Notification protocol faces a downside in cases of crowding. Nodes are forced to connect beyond the available bandwidth in crowded areas. A solution to this would be to decrease the amount of connections by reducing the player's area of interest. Another solution would be to route messages through another node which has sufficient bandwidth.

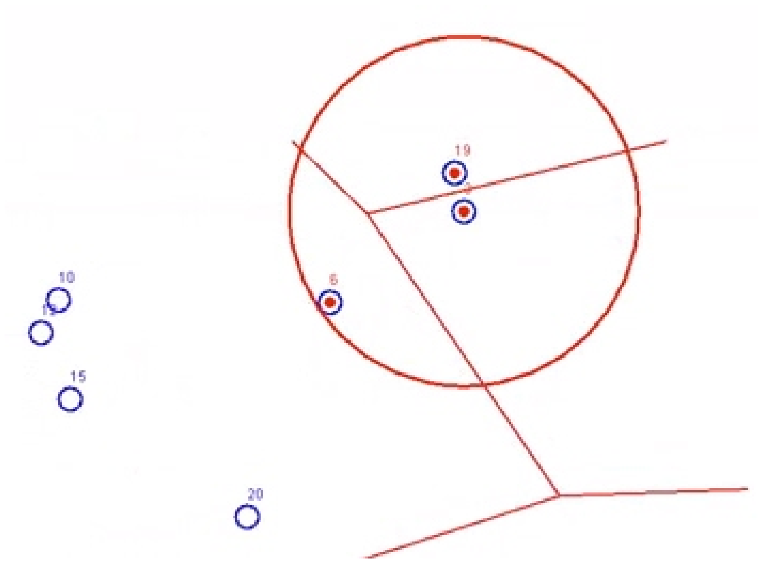


Figure 2.5: An example of mutual notification taken from the Hu *et al.* [2006] demo.

2.4.4 Evaluation of Protocols

An evaluation conducted by Krause [2008], relating to the performance of the peer-to-peer protocols discussed in this section, revealed the following: The Mutual Notification protocol was shown to have the best performance in minimizing message delays and the delay rate did not change as the test group size got larger. It was also shown that the bandwidth requirements for exchanging messages were moderate as long as a certain location was not densely populated. The Application Layer Multicast protocol initially displayed results of reasonable delay and bandwidth requirements, but as the test group size got larger the protocol did not cope well. It displayed attributes of high bandwidth consumption and unacceptable message delays. The Supernode Control protocol did not suffer from unmanageable message delays as the group size increased, though like the Application Layer Multicast, it did show that a densely populated area will require more bandwidth. This was most likely due to the static division of the game world and poor load distribution with the system tested. Although the Mutual Notification protocol had the best performance, the protocol does not contain a "dominant" node to resolve situations when nodes calculate different game states. If a single player were to compute a different game state this would lead to a "butterfly effect"[6] across the game.

2.5 Hybrid Gaming Networks

A combined approach of mixing peer-to-peer and client-server architectures is also possible. Negatives in the peer-to-peer system can be alleviated by allowing specific operations to be handled by a server-based implementation. For example, in peer-to-peer gaming networks the complexity of cheat detection and prevention is one of the major reasons game developers avoid peer-to-peer architectures [Yahyavi and Kemme 2013]. Therefore, a trustworthy server can be used to overlook the players' actions in a peer-to-peer network to ensure that there is no player cheating. Yahyavi and Kemme [2013] list three possible combinations of client-server and peer-to-peer networks:

[6]The butterfly effect is a notion in chaos theory, where a small but sensitive change in a system (be it an ecological or a computing system) causes a major (often undesired) change in later stages.

- Cooperative message dissemination: The game state is maintained by the server but messages containing the state of the game are sent using multicast mechanisms constructed by the peer nodes. This approach reduces the bandwidth requirements of the server.
- State distribution: The game state is distributed amongst the peers, and thus the peers are responsible for the game functions. The server manages how peers communicate with each other. The server is also responsible for centralized operations such as player authentication and keeping track of players joining or leaving the game. This model achieves scalability by distributing the game state amongst several nodes.
- Basic server control: This approach sets the messaging propagation and state distribution amongst the peers. The server only maintains sensitive information such as: user login credentials, payment information and the players' progress. The server also handles the process of a player joining or leaving a game.

OnDeGas (On Demand Gaming Service) is a hybrid network gaming architecture designed to solve the scalability issue within client-based systems and latency within peer-to-peer architectures [Barri *et al.* 2010]. *OnDeGas* consists of a master server that carries out all the game functions. If a region in the game world becomes crowded, slowing down the operation of the whole game, the region is zoned off and handed to a pre-selected peer that becomes the supernode of that region. The master server also coordinates region selection, supernode nomination, message communication methods between supernodes and bootstrapping. Barri *et al.* [2010] do tackle the issue of fault-tolerance by replicating the supernodes data to another node. If the supernode were to fail the node containing the replicated data can take over as the new supernode. Barri *et al.* [2010] failed to examine or discuss if the game would be operable if the master server failed. This likely means that the master server is the single point of failure for the whole network architecture.

2.6 Conclusion

Modern MMOG use client-server models to provide online gaming. Significant development has gone into ensuring the network architecture provides a highly interactive game environment without any delay. The techniques used to decrease latency delay range from dead reckoning, delta compression, interest management, and most importantly, transport layer selection. However, scalability within client-server architectures is limited and costly. We argue that peer-to-peer architectures provide better support for MMOG by allowing the user base to grow as large as possible. Peer-to-peer gaming architectures produce several challenges that must be addressed. These challenges are game state management, scalability, fault-tolerance, delay management, message dissemination and cheating. Game state management addresses concerns with game state consistency and data persistence in a distributed environment. With respect to delay management, the peer-to-peer application must process game play actions and synchronise game state immediately in order to present an interactive game with no game play delays. Furthermore, communication delays between hosts in the gaming network need to be kept to a minimum. We also highlighted the performance of existing message dissemination protocols for peer-to-peer architectures. The Application Layer Multicast protocol has moderate delays and bandwidth requirements which do not complement peer-to-peer gaming networks; therefore, the Application Layer Multicast protocol is eliminated from being used in this research, leaving the Supernode Control protocol and Mutual Notification protocol as our remaining choices to use for message dissemination. The chosen protocol is based on our peer-to-peer gaming solution, discussed further in Chapter 3. Prevention against cheating is another desired feature for peer-to-peer gaming architectures. However, the focus of this research is on the delay management of peer-to-peer gaming architectures. For this reason, the aspect of cheating is left for future work. The next chapter outlines the research questions, the aim of this research and research methodology.

Chapter 3

Research Questions and Methodology

3.1 Introduction

In Chapter 2, we discussed the background of this research and the issues facing the design, implementation, and deployment of massively distributed online games. In this chapter, we present the methodology used to conduct this research. In particular, the research question described in Chapter 1 is formalized in Section 3.2 along with a subset of research questions. The aim of the research is presented in Section 3.3. Subsequently, we provide an outline of the methodology that is used to answer the research questions in Section 3.4. Specifically, we discuss the concept behind our peer-to-peer solution. The design concept is related back to the research question. We conclude this chapter by formulating a number of tests to answer the research questions.

3.2 Research Questions

The client-server architecture, where game execution and management is controlled by a dedicated server, is currently the prevalent architecture for online gaming. However, client-server systems present problems such as high server costs and limited user limits. Moreover, the server represents the single point of failure. Peer-to-peer networks by nature are robust against component failure, reduce the cost of deploying servers, and are highly scalable. Peer-to-peer architectures are therefore an attractive field to host MMOG. The central question that we seek to address in this research is:

What is the best approach to reliably and timely deliver messages from one host to another in a peer-to-peer architecture?

In order to address the central question, we also have to answer the following questions:

- *How to deliver messages in a timely manner within a peer-to-peer network?* From a communication point of view, the latency problem is related to network delays experienced during the transportation of data over the network. In Section 2.2.3 of Chapter 2, we noted the choice in the transport layer protocol is dependent on the type of game. Despite TCP's reliable delivery and network congestion prevention capabilities, UDP is used for action-based games. The reason UDP is preferred over TCP in action-based games is due to the TCP property of packet arrival order which may delay the delivery of messages to the application layer. Though the delays experienced using TCP are undesired in "real time" applications, such as VoIP and IPTV, media streaming applications still seek the services which TCP provides. Protocols such as RTP (Real-time Transport Protocol) have been developed to offer TCP services but on top of the UDP protocol [Schulzrinne *et al.* 2003]. Despite the availability of such protocols most media streaming applications make use of TCP, because data delivery for clients using the Transmission Control Protocol is less complicated over traversed networks (such as proxies and firewalls) [Harcsik *et al.* 2007]. Also, TCP's flow control service avoids congesting the network by adapting the rate of transferring messages dependent on the load conditions of the network [Kurose and Ross 2009]. There are other transport layer protocols which have been developed to improve on the Transmission Control Protocol delay, such as the Stream Control Transmission Protocol (SCTP) [Stewart *et al.* 2006]. However, these transport layer protocols require a change in the transport-layer protocol stack, which makes SCTP and other newly developed transport layer protocols difficult to deploy across existing routers [Dukkipati *et al.* 2010; Google 2012]. Given the difficulty of improving transport layer protocols without negatively effecting communication between nodes, the application layer must be refined to reduce TCP delay. Taking from the ideas above, we seek to use TCP in a different manner, which will allow messages to be delivered under a specific amount of time.

- *Given the distributed nature of the system, how can the information about the game state be kept consistent among the players (nodes), whilst guaranteeing availability of the system?* In Section 2.3 of Chapter 2, we identified game state management as one of the challenges facing peer-to-peer gaming systems. Firstly, peer-to-peer gaming systems need to present a consistent state to the players. Given the scalability factor of peer-to-peer systems, mechanism are required to distribute the data amongst the nodes. With respect to nodes abruptly leaving the network, a peer-to-peer gaming system must ensure that the game state data is always accessible. Also, given the constant changes that may occur in the game state, the peer-to-peer system needs to be highly available to perform read and write operations on the game state data. We seek a highly available game data management model that can maintain the game state in a distributed system.

- *What concurrent programming paradigm can reduce the time taken to process numerous messages retrieved during network exchanges?* In Section 2.2 of Chapter 2, we discussed delay management techniques within MMOG. We highlighted the methods used to reduce *synchronization delays* within online games, for example dead reckoning. Another delay factor for online games resides in the time taken to process a message packet received from another host. Hosts within a game network exchange a number of packets, indicating actions taken or a change in the game state, which a games application layer needs to process [Fiedler 2008; Valve 2012]. Given a large user base, the number of message packets a server receives grows proportionally. In a MMOG, the server is required to concurrently process a large number of incoming messages without delaying game interactivity. We seek a highly concurrent model to process the network messages and reply promptly to request messages.

3.3 Research Aim

The purpose of this research is to create a peer-to-peer system for hosting MMOG. The architecture was designed to satisfy the following criteria:

- Consistency
  MMOG operate on shared game states. The game states must be consistent between all players. Consistency allows players to perceive events identically.

- Availability
  The system must be available to meet all write and read requests with a response.

- Fault-tolerance
  In distributed systems, a node may unexpectedly fail. The proposed system will be tolerant of nodes dropping out at any point with no disruption to game play.

- Scalability
  The architecture will be designed to handle a large number of concurrent users in the same game world without experiencing loss to the quality of service.

- Low latency
  Many games (such as action-based games) require that messages arrive under a certain time or the user will experience lag. Therefore the time for a message to arrive at another node must be kept below a threshold pertaining to the game.

3.4 Research Methodology

The research was completed by creating a distributed application solution that would be used for MMOG. Aside from the networking application being specifically directed to MMOG, the network application has the potential to be used for other applications such as Voice over Internet Protocol (VoIP), media streaming, and instant messaging. The concept behind the design approach of the system was to separate the business logic away from the networking. The network is designed for general online gaming and is not constrained to meet specific requirements for a certain game. Rather the distributed gaming network provides a set list of features, which if it satisfies the requirements for a game, can be used for an online multiplayer game. Figure 3.1 gives the high level schematic design behind the distributed network solution. The architecture adopts the design of the Supernode Control protocol, where a supernode co-ordinates the message dissemination, game execution and state management of a selected region. The Supernode Control protocol is able deliver messages in an adequate time frame, as noted in Section 2.4 of Chapter 2. Unlike the Mutual Notification protocol, that does not ensure that a local group of players have a consistent view, the Supernode Control protocol warrants that players based in a specific region have a consistent view.

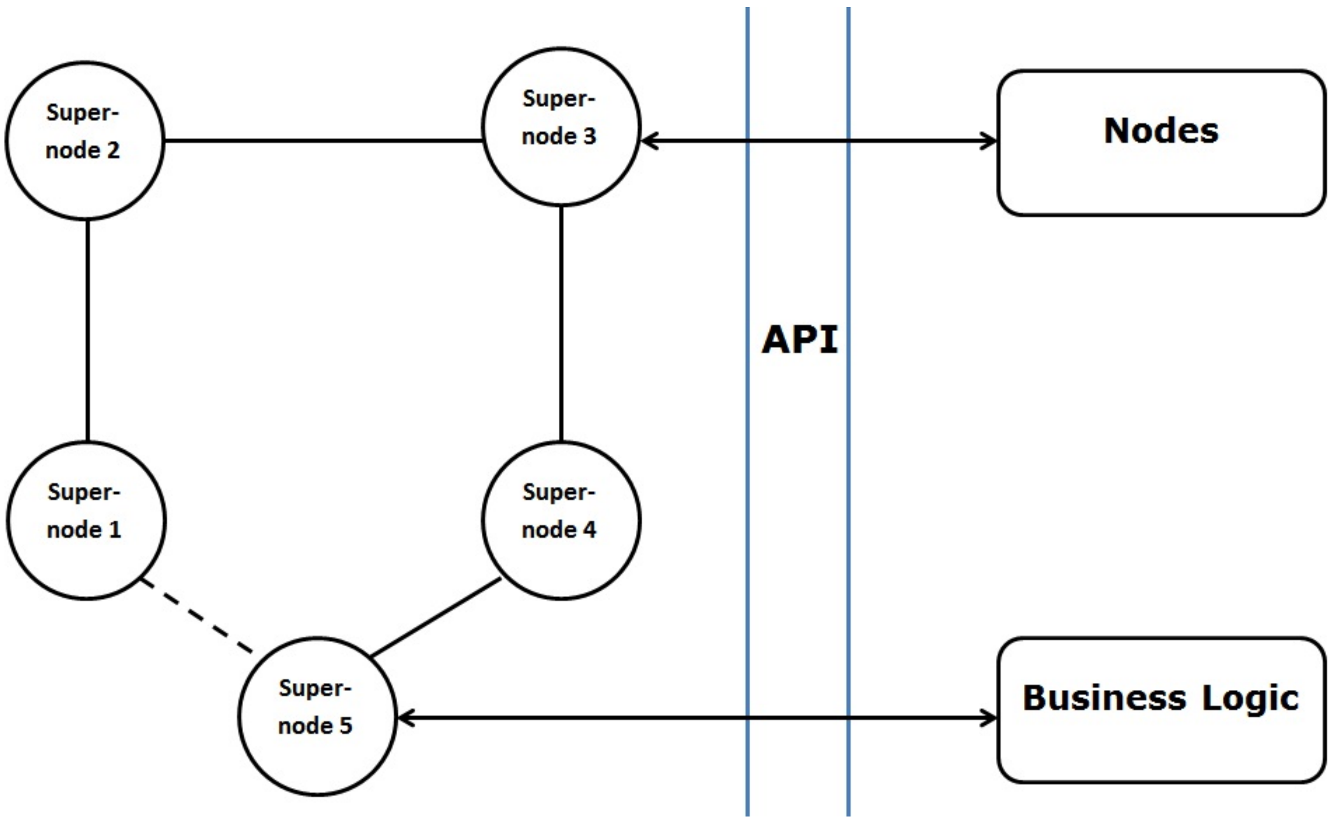


Figure 3.1: An abstract idea of the distributed network.

Each node in the distributed solution offers three utilities that nodes in the system interact with through an Application Programming Interface (API). The three utilities offered are a distributed storage system, a messaging service and a routing service that enables hosts to interact with services outside the distributed network application. The services were identified from issues facing distributed games and services required in distributed gaming networks.

The storage service provides a distributed storage where game state (and other data) can be stored. The second utility is a publish and subscribe service that allows neighbouring players or players in a specific zone (dependent on the message dissemination protocol) to subscribe to a host and receive relevant messages like game state changes. The third offering utility is a service routing component. Within the business logic, there may be other services which the business logic contains which the network application does not provide. The service routing component coordinates requests between hosts and the services. For example, if a node is nominated as a supernode for a zone, the supernode is required to calculate the game state after receiving players' actions. The supernode could route the actions to the logical service that computes the game state. After the game state has been determined, it can be broadcast to the players. In Chapter 5, the design of the system developed is discussed in detail.

Validation of Research

In light of the questions discussed in Section 3.2, the validation methodology consists of experiments to measure the latency of the peer-to-peer solution. To answer the research question, we measure the performance of the communication model used to send messages between nodes. The results determine if it is possible guarantee reliability and deliver messages in a timely manner. We also examine the time taken for services provided by the nodes to respond to requests. The experiment provides us with insight into the chosen model of concurrency performance. Furthermore, we compare the front-end service of our peer-to-peer solution against web servers with the capabilities of hosting online games. The experiment outlines a benchmark of our systems performance against existing online gaming solutions. More importantly, we examine the response time of our solutions functions in a distributed environment, in order to inspect if the system is able to scale as more supernodes are introduced.

3.5 Conclusion

In this chapter, we formalized the research questions and presented the design overview of our peer-to-peer solution. Our solution builds on solutions from various domains. These include communication protocols, distributed data systems and models of concurrency. In the following chapter, we present these building blocks for our peer-to-peer gaming solution.

Chapter 4

Building Blocks

4.1 Introduction

Online games present a set of requirements that need to be satisfied to host a game in a peer-to-peer infrastructure. These requirements include managing game state consistency, reducing the delay of sending and processing a messages, and techniques to handle cheating amongst players. In light of the research questions discussed in Chapter 3, we investigate related approaches to the challenges facing peer-to-peer gaming networks. The investigation presented in this chapter coincides with the solutions used within our peer-to-peer gaming network. In particular, we investigate network delay management within web applications, synchronization delay management with regards to processing network messages in a timely manner, and data management within distributed database systems. Section 4.2 presents communication protocols designed to reduce web page load times. Section 4.3 illustrates the challenges of guaranteeing consistency, concurrency and scalability in distributed database systems. We also explore distributed database models created from the growth of web data. In Section 4.4, we discuss highly concurrent programming paradigms that can be used to process numerous network messages.

4.2 Communication Protocols

In Section 3.2, we highlighted the research question, which is to seek a communication mechanism to reliably and timely deliver messages between hosts in a peer-to-peer gaming network. We proposed the use of TCP to deliver messages between hosts, because the TCP transport protocol can easily traverse networks, guarantee message delivery, and control the flow rate of messages [Harcsik *et al.* 2007]. However, the message delivery guarantee provided by TCP may cause undesired latency. Therefore, we seek to refine the TCP communication model to mitigate the retransmission delay. Web applications exemplify applications that use TCP for communication, but also seek to decrease the time taken to exchange content. In this section, we present application layer protocols that refine communication models on the web. Initially we present the HTTP application layer protocol used to transfer data. We then present two application protocols, WebSocket and SPDY, that have been developed to improve the communication model between web applications.

4.2.1 Hypertext Transfer Protocol (HTTP)

The World Wide Web (abbreviated as WWW, commonly known as the Web), is undoubtedly one of the most used applications on the Internet. The Web contains information and objects of various mediums [Kurose and Ross 2009]. These objects range from HTML files, images, video clips, video games, and more. Information and objects are often accessed from sources called web pages documents, that are addressable by a single URL [Kurose and Ross 2009]. The Hypertext Transfer Protocol (HTTP) is the Web's application layer protocol. HTTP is defined as an application-level protocol for distributed, collaborative, hypermedia information systems which is run on top of TCP [Fielding *et al.* 1999]. HTTP is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [Fielding *et al.* 1999]. A popular feature of HTTP is the typing and negotiation of data representation, which allows systems to be built independently of the data being transferred. Hypertext Transfer Protocol Secure (HTTPS) is a secure communication protocol for exchanging HTTP messages. HTTPS is layered on top of Secure Sockets Layer (SSL) or Transport Layer Security (TLS) [Dierks and Rescorla 2008; Freier *et al.* 2011], with SSL/TLS lying in the session layer of the OSI model. Secure communication over the Web was required to prevent eavesdropping and middleman attacks between hosts exchanging sensitive information [Kurose and Ross 2009].

The first documented version of HTTP was in 1991, with HTTP 0.9. The second was in 1996 with HTTP 1.0 [Berners-Lee *et al.* 1996]. The last revision of HTTP was made in 1999, HTTP 1.1, which is still widely used. However, there has been consideration that this web protocol has become increasingly out-of-date with the networking and computing resources used today [Fette and Melnikov 2011]. Over the past years, web applications have become far more complex that a standard HTTP system could not provide the services required by some applications. Below, advancements made in web application protocols from the weaknesses in HTTP 1.1[1] are discussed.

4.2.2 WebSocket

A need for a simpler two-way communication between hosts came from the issue faced by certain web applications that used bidirectional communication between the client and the server. When a browser visits a web page, an HTTP request is sent to the web server. The web server would acknowledge the request and respond with the requested web page. In many cases, the information contained on the web page could be old by the time the web page is rendered. For example, stock exchange prices, news reports, and traffic information are constantly changing. In order to obtain the latest information, the user would have to refresh the web page manually. A better solution would be to have the web server push down the new information to the web browser automatically.

Unfortunately, HTTP does not support full-duplex communication naturally. Web applications that used bidirectional communication required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls [Fette and Melnikov 2011]. This resulted in a number of problems such as a sever being forced to use a number of different TCP connections, one for sending information to the client and a new one for each incoming message. Each client-to-server message would have an HTTP header, creating a high overhead. Also, the client application would be forced to maintain a mapping from the outgoing connections to the incoming connection, in order to track replies.

As part of the HTML5 initiative, the WebSocket protocol was created to supersede existing bidirectional communication technologies (such as Comet [Crane and McCarthy 2008]) that use HTTP. The WebSocket protocol enables full-duplex communication between two hosts over TCP through a single socket [Wang *et al.* 2012]. The WebSocket protocol is an independent TCP protocol, with its only relationship to HTTP being its handshake that is interpreted by HTTP servers as an upgrade request. The benefits from the HTTP handshake include the ability to traverse firewalls, proxies and authentication servers. WebSockets also place less of a burden on servers, allowing existing machines to support more concurrent connections.

4.2.3 SPDY

SPDY is an application protocol developed by Google to decrease web page load times [Google 2012]. SPDY was created to address the performance issues inherent in HTTP including HTTP's lack of pipe lining and message prioritization, the inability to send compressed headers and the absence of server push capabilities [Thomas *et al.* 2012]. Belshe and Peon [2012] engineered SPDY to run over a SSL connection, introducing a connection latency penalty, in the belief that the web should be secured by

[1]The Internet Engineering Task Force (IETF) is currently working on the next version of HTTP, HTTP 2.0 [Belshe *et al.* 2013]

default. SPDY does not replace HTTP, but modifies the way HTTP requests and responses are sent. This allows content on servers to remain unchanged. The SPDY protocol is made up of a session layer on top of a SSL, which facilitates multiple concurrent and interleaved streams over a TCP connection. Figure 4.1 shows the SPDY protocol stack.

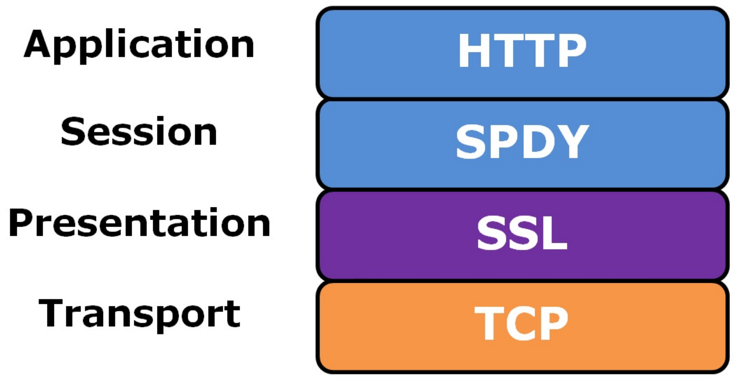


Figure 4.1: The SPDY protocol stack, adapted from Google [2012].

SPDY has been implemented in several web servers, browsers and libraries including the Jetty Web Server, Apache webserver, Netty library, Google Chrome and Mozilla Firefox. SPDY has also been used on Google websites, Facebook and Twitter. The SPDY protocol is also being used as the base for HTTP 2.0 [Belshe *et al.* 2013]. SPDY uses the following three features to improve the web page load time [Belshe and Peon 2012]:

i Multiplexed streams: SPDY allows for unlimited concurrent streams over a single TCP connection. Because requests are interleaved on a single channel, the efficiency of TCP is much higher: fewer network connections need to be made, and fewer, but more densely packed, packets are issued.

ii Request prioritization: A side-effect of multiplexed streams is congestion. If the bandwidth of a channel is constrained due to multiple requests, critical requests might be delivered more slowly. To solve this issue, SPDY implements request prioritization in which a client can assign priorities to each request. This prevents the network channel from being congested with non-critical resources when a high priority request is pending.

iii HTTP header compression: SPDY compresses request and response HTTP headers in order to decrease the number of packets and bytes transmitted.

Apart from these three features, SDPY also offers two more advanced features which can be used by a server to accelerate content delivery:

i Server push: SPDY allows a server to push resources to a client which the client might request. The data is kept in the client's local cache and upon needing the data the client can retrieve it from there.

ii Server hint: Rather than forcibly pushing data onto the client, the server can rather send a suggestion on the resource(s) that is required. This allows the client to be involved in making a decision if the resource should be retrieved from the server. For example, the client could check its local cache, requesting the resource if it does not reside in the client's cache.

Tests conducted by Google have shown SPDY to have a 27% - 60% speedup when downloading webpages on a plain TCP connection (without SSL), and a 39% - 55% speedup over a SSL connection [Google 2012]. SPDY's header compression was one of the attributes which led to the improvement of download time. The header compression resulted in the request headers being reduced by ~88% and the response headers being reduced by ~85%. On lower bandwidth lines, the header compression reduced page load times by 45 - 1142 ms. Further tests were conducted to examine the effect of packet loss on SPDY's timing on delivering packets versus that of HTTP. During packet loss tests, SPDY latency savings proportionally increased up to 48% at the 2% packet loss rate. The increase stopped at the 2.5% packet loss rate with a 44% speedup. The improvements in content delivery against packet loss were attributed to following features in SPDY: SPDY approximately sends 40% fewer packets than HTTP, meaning fewer packets are affected by loss. SPDY's more efficient use of TCP triggers TCP's fast retransmit instead of using retransmit timers. Given that SPDY uses fewer TCP connections, the chance of losing a *SYN* packet decreases. The delay caused by the loss of a *SYN* packet is extremely expensive (up to a 3 second delay) [Google 2012]. Another test examining the improvement on the Round Trip Time (RTT) was also conducted. SPDY's latency savings also increased proportionally with increases in RTTs, up to a 27% speedup at 200 ms. The improvement on the RTT was due to SPDY being able to fetch all the requests in parallel. If a HTTP client has 4 connections per domain, and 20 resources to fetch, it would take roughly 5 round trips to fetch all 20 items. SPDY fetches all 20 resources in a single round trip. Detailed results for the above mentioned tests can be found in Appendix B.

4.2.4 Summary

UDP may seem like the appropriate choice to deliver messages in a timely manner. However, UDP does not provide any form of message delivery guarantee, has no congestion avoidance mechanism, and may be difficult to traverse networks. Improving on the transport-layer is risky because devices such as routers may not be able to use the new protocol, as is the case with SCTP. By using SPDY as the communication model between nodes, we seek to use TCP in a different manner that would reliably deliver messages under a specific time. SPDY's multiplexing property allows multiple requests/responses to be sent over a single channel, diminishing the delay experienced by TCP when packets are dropped. SPDY also incorporates the advantage of HTTP to traverse proxies and firewalls.

4.3 Distributed Data Systems

Peer-to-peer online gaming networks face the challenge of maintaining a consistent game state across several hosts. In this section we explore the properties of consistency, availability and data persistence in distributed data storage systems. Section 4.3.1 presents the challenges of guaranteeing consistency, scalability and availability in a distributed system. In Section 4.3.2, we introduce the two most common distributed database systems, relational and non-relational database systems. In particular, we examine the scalability limitation in relational database system. Finally, Section 4.3.3 concludes the section by discussing a low latency distributed storage system, Voldemort. We outline some of internal concepts of the Voldemort distributed database system and describe how Voldemort handles replication and partitioning.

4.3.1 Dilemmas Facing Distributed Database Systems

A trivial method of storing data is on a single high performance machine. However, as with client-server architectures, this methodology has a number of drawbacks. The machine can only serve a number of read and write requests, often attributed to the hardware limitations of the machine. The machine hardware specifications can be updated but this is costly and may lead to downtime as the upgrade is being performed. Furthermore, the machine is the single point of failure. Distributed data storage systems have become a prominent approach to address the above issues. Similar to peer-to-peer gaming networks, distributed data systems have a number of requirements and challenges. The growth of web storage application drove a rise in distributed systems that led to the correlation between availability, consistency and scalability within distributed management systems.

The CAP theorem was a conjecture presented by Brewer [2000], which was later formally defined and proven by Gilbert and Lynch [2002]. Large distributed database systems arouse issues and properties that small data systems do not have to worry about. Brewer [2000] identified three distinct properties which are desirable in distributed data systems:

- **C**onsistency: All nodes in the distributed system have the same data; that is, a consistent transaction is one that starts with a database in a consistent state and ends with the database in a consistent state. For example, if a data value (say x) was added/updated, all the nodes in the system would contain the same value for x.
- **A**vailability: The system must guarantee that every read or write request is fulfilled.
- **P**artition-tolerance: A distributed system must provide some amount of fault-tolerance. This property indicates that the system should operate despite a node failing.

From the above three properties, the CAP theorem states: *Though it is desirable to have Consistency, High-Availability and Partition-tolerance in every system, unfortunately no system can achieve all three at the same time.* Figure 4.2 shows the properties which a distributed system will guarantee according to the CAP theorem. Only two of the three CAP properties can be guaranteed in a distributed database system. The three distinct combinations which are achievable in a distributed database system are discussed below.

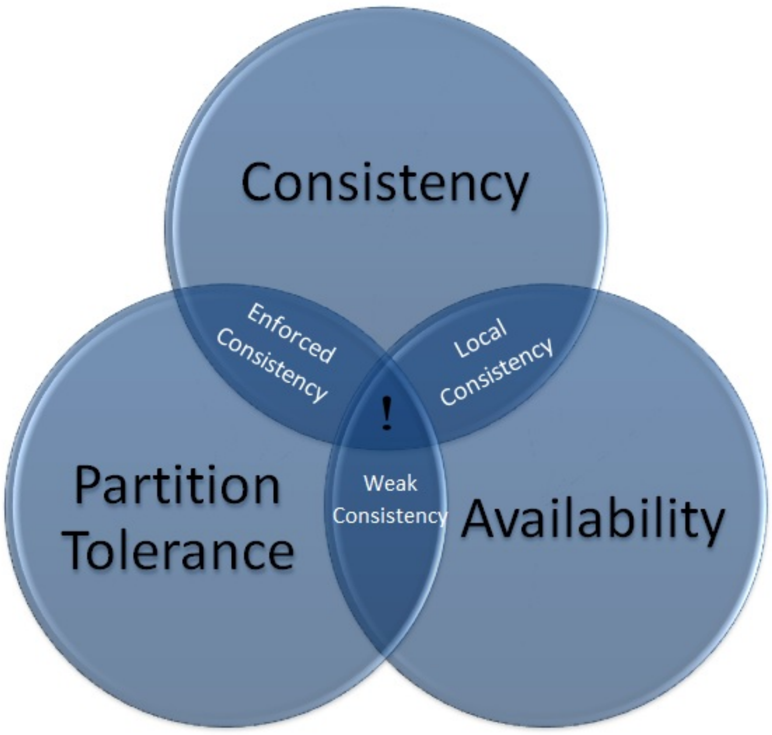


Figure 4.2: The different properties that a distributed system can guarantee based on the CAP theorem.

Consistency and Availability without Partition-tolerance (local consistency)

A system that is not tolerant to network partitions can satisfy the availability and consistency properties. This means the system has to be in one environment (a single machine or rack) for these properties to hold. A drawback of this data storage model is that it does not permit scaling. It also creates a single point of failure. Enforcing local consistency goes against the nature of distributed data systems as the data cannot be distributed amongst several nodes. In order to obtain a large distributed-scale system, network partitions are given; therefore consistency or availability must be dropped [Vogels 2009].

Consistency and Partition-tolerance without Availability (enforced consistency)

A system that employs consistency and partition-tolerance provides enforced consistency amongst the distributed nodes, which means that at all times every node must have the same value for data contained within the distributed system. However, employing enforced consistency decreases the availability of the system to respond to requests. When a value is modified, all nodes in the system must conform to this new value. Until all the nodes have a consistent state, the system may not be able to respond to requests. Maintaining a consistent state is fairly complex when there are many nodes, especially in the case of a partition failure. Paxos protocols introduce methods in maintaining consistency amongst nodes [Lamport 1998].

Availability and Partition-tolerance without Consistency (weak consistency)

Systems that are always available and are tolerant to partitions do so by dropping immediate global consistency. This allows a system to be readily available to requests, though responses may not contain the correct value. Highly available and partition-tolerant systems employ a best-effort consistency technique. Instead of enforcing consistency amongst nodes, updates are steadily spread across the nodes until all the nodes contain the latest updated value. This is referred to as *eventual consistency*, a specific form of weak consistency [Vogels 2009]. Eventual consistency guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. The Domain Name System (DNS) is an example of a system that implements eventual consistency.

4.3.2 Types of Distributed Database Systems

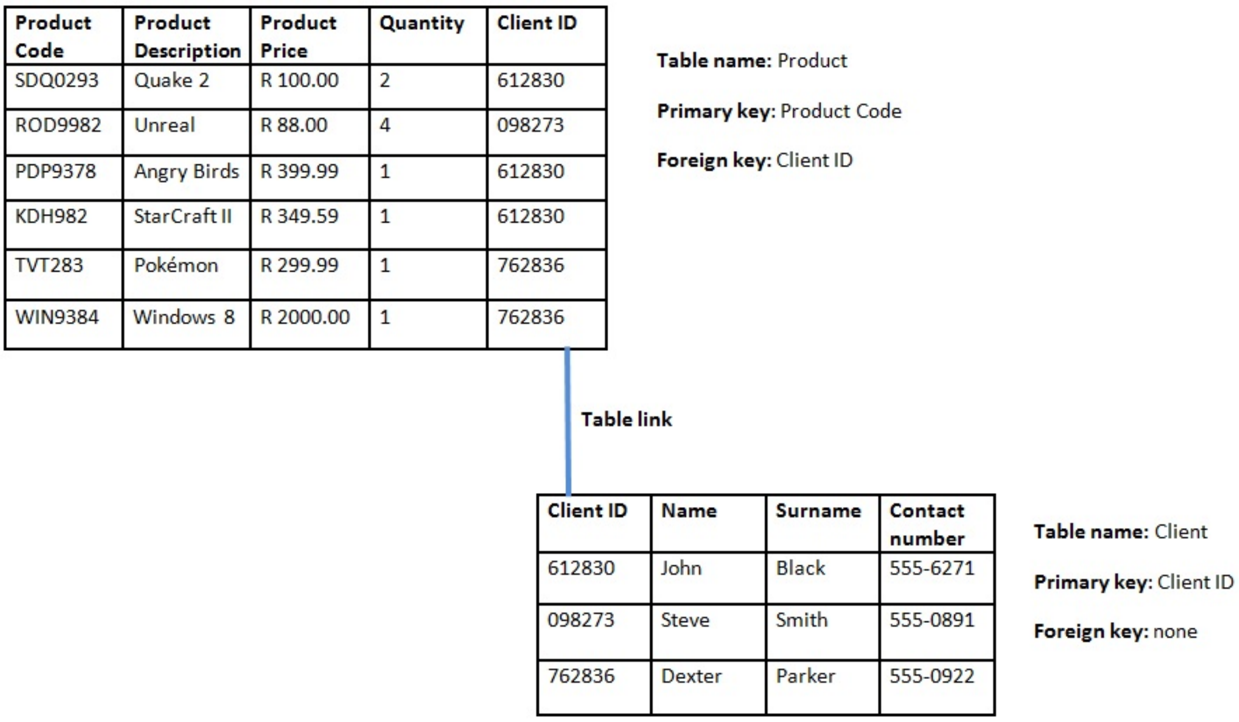
This section discusses two prominent database models, relational and non-relational database systems. In particular, we look at the implications of increasing web data that has caused the move from relational to non-relational database systems. Furthermore, we take a brief overview of the different types of non-relational database systems.

Relational Database Management Systems

The most common model for storing data is based on the relational model, formulated and introduced by Codd [1979] in the seventies. In the relational model data is stored in a tuple, forming an ordered set of attributes. A relational database is a collection of data items organized as a table. The table columns are referred to as attributes and the rows are called tuples, making a relational database table an organized group of relational model items. Tuples in different tables can be referenced by using a foreign key, see Figure 4.3. A foreign key is an attribute in a table whose value can match a unique primary key in a related table. Structured Query Language (SQL) has become the standard programming language for interfacing with relational database systems. Relational database management systems that use SQL include Oracle Database, Microsoft SQL Server and MySQL.

Relational databases became the dominant storage model for a number of reasons. Transactional mechanisms were employed to coordinate access to data and guarantee consistency. The relational model also offered integration with different applications; that is, different applications could access the same database system [Sadalage and Fowler 2013]. One of the major drawback for using a Relational Database Management System (RDBMS) is that it requires substantial hardware and system software [Rob *et al.* 2007]. They also run on a single node, making the relational database system the single point of failure for any application dependent on the data. Relational databases provide solid, well-developed services that adhere to the Atomicity, Consistency, Isolation, Durability (ACID) properties. But the implications of the ACID paradigm make it difficult to build a distributed database system based on the relational model [Erb 2012]. Enforcing the ACID properties on a distributed database requires complex mechanisms that restrain low latency and high availability.

Relational database systems remained the standard storage model for many years despite research being conducted into other forms of storage. This was due to the RDBMS integration property. An organization with several applications interacting with the relational database would have to change the interface of each application if the database model were to change. This drawback hindered other database models being implemented. In the early 2000s as the web began to grow, popular websites



Product Code	Product Description	Product Price	Quantity	Client ID
SDQ0293	Quake 2	R 100.00	2	612830
ROD9982	Unreal	R 88.00	4	098273
PDP9378	Angry Birds	R 399.99	1	612830
KDH982	StarCraft II	R 349.59	1	612830
TVT283	Pokémon	R 299.99	1	762836
WIN9384	Windows 8	R 2000.00	1	762836

Table name: Product

Primary key: Product Code

Foreign key: Client ID

Client ID	Name	Surname	Contact number
612830	John	Black	555-6271
098273	Steve	Smith	555-0891
762836	Dexter	Parker	555-0922

Table name: Client

Primary key: Client ID

Foreign key: none

Figure 4.3: Simple relational database tables linked by an attribute.

received large amounts of traffic. This in turn meant that the data stored by websites grew proportionally. For example, online shopping web sites needed larger databases to store new customer information. The probable solution was to scale up the number of machines serving the users. Scaling a relational database is costly as an RDBMS requires substantial amount of hardware to operate. Furthermore, scaling up RDBMS would employ "unnatural acts" that would often cause loss in querying functions, referential integrity, transaction and consistency control [Sadalage and Fowler 2013].

NoSQL Systems (Non-Relational Database Management Systems)

The mismatch between relational database systems and scalability led to development in simple and flexible non-relational database systems. A solution to scaling up a database was to use a cluster of commodity machines to handle the increasing user loads. This was more cost efficient than employing a new powerful machine every time the user rate grew. Web-oriented companies such as Google and Amazon began developing and publishing papers on non-relational database models that were used on a large cluster of machines [Chang *et al.* 2006; DeCandia *et al.* 2007]. As several institutions also developed non-relational databases, the term NoSQL was devised to describe distributed systems that are designed to scale and are highly available. In context of the CAP theorem, NoSQL storage systems trade consistency in favour of availability and partition-tolerance. There is no formal definition for a NoSQL system, though Sadalage and Fowler [2013] notes that database systems with the following properties commonly qualify as NoSQL systems:

- Not using the relational model: The system does not use SQL language, nor does it try to satisfy the ACID properties.
- Designed to run on large clusters
- Designed to accommodate the storage needs for 21st century web applications.
- No schema: Relational data can only be stored in a database if it meets the schema of that database. NoSQL systems do not use a schema, allowing fields to be added without having to define a structure first and increasing the flexibility of migrating data.

- Open source: Though there exists NoSQL database systems that are closed-source, the phenomenal growth in NoSQL research lies with open-source systems.

There exists a wide range of NoSQL systems, each driven by a specific database model. A variety of the models often overlap on their design. Below we list the four basic classifications on NoSQL systems.

- **Key/Value Stores**

  Key-value databases are the simplest NoSQL data stores based on hash tables. Data is stored in a tuple format with a unique key and value. The value is an arbitrary chunk of data that is indexed and queried by the unique key. The simplicity of the key-value data model provides scalability and performance. However, query opportunities are generally limited as the database only uses keys for indexing [Erb 2012]. Some prominent key-value storage systems include Amazon's Dynamo [DeCandia *et al.* 2007], Redis [Carlson 2013], Riak [Fink 2012] and Project Voldemort (which we discuss in detail in Section 4.3.3).

- **Document Stores**

  Document store databases are related to key-value data stores. A single document contains a structured collection of key-value items such as JSON, XML and BSON. Against key-value stores, document stores allow for more complex queries; therefore, a document can be used for indexing and querying other values [Erb 2012]. Apache CouchDB [Anderson *et al.* 2010], MongoDB [Plugge *et al.* 2010] and RavenDB [Ritchie 2013] are some of the most used document store databases.

- **Column Stores**

  Column data stores are based on Google's *Bigtable*; a sparse, distributed, persistent multi-dimensional sorted map [Chang *et al.* 2006]. *Bigtable* handles petabytes of data across thousands of commodity servers that are used for web indexing, MapReduce, Google Earth, Google Maps, YouTube, Gmail and several other Google web projects. A map is indexed by a row key, column key. Column store maps are characterized by row keys that are mapped to column families (see Figure 4.4). Bigtable's map is indexed by sorted row keys, column keys and timestamps, thus making it a three dimensional map. Other common column family stores include Amazon SimpleDB [Habeeb 2010], Apache Cassandra [Lakshman and Malik 2010] and Apache HBase [Konishetty *et al.* 2012].

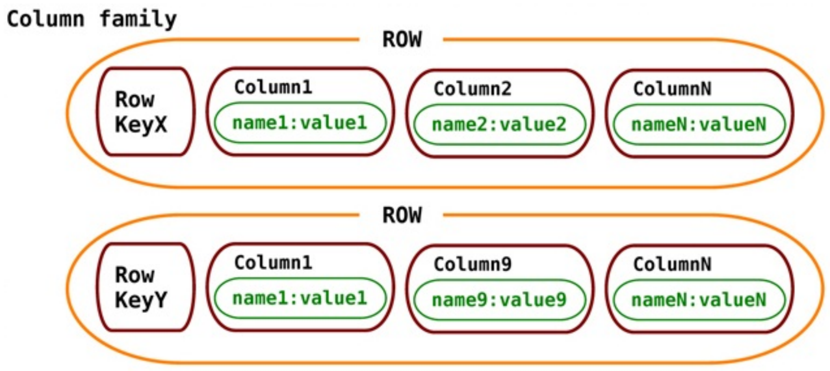


Figure 4.4: Column store map based on the Apache Cassandra model [Sadalage and Fowler 2013].

- **Graph Stores**

  Graph databases, based on graph theory, introduce the notion of relationships between nodes. Nodes represent entities containing properties (data) and the edges represent a relationship between the nodes. The nodes are organized by relationships, revealing patterns between nodes. For example, Figure 4.5 illustrates the relationship between entities in a social network. The graph can be queried in a number of ways, such as finding out who is employed by Big Co. and likes NoSQL Distilled. This information can be used to make a number of suggestions to the users (nodes), amongst several other applications. A query made on a graph is also known as *traversing* the graph. The requirements used to traverse a graph can be changed without changing the nodes or edges of a graph. A notable benefit to using graph databases is the efficiency in traversing the relationships. The relationship between nodes is not calculated at query time but is persisted. That is, the organization of the graph is stored and interpreted in different ways based on the relationships. Nodes may have several different relationships between them. This allows for direct relationships between nodes and secondary relationships that are used for paths, time-trees, quad-trees for spatial indexing, or linked lists for sorted access [Sadalage and Fowler 2013]. Some well-known graph databases are Neo4j [Webber 2012] and InfiniteGraph [van der Lans 2010].

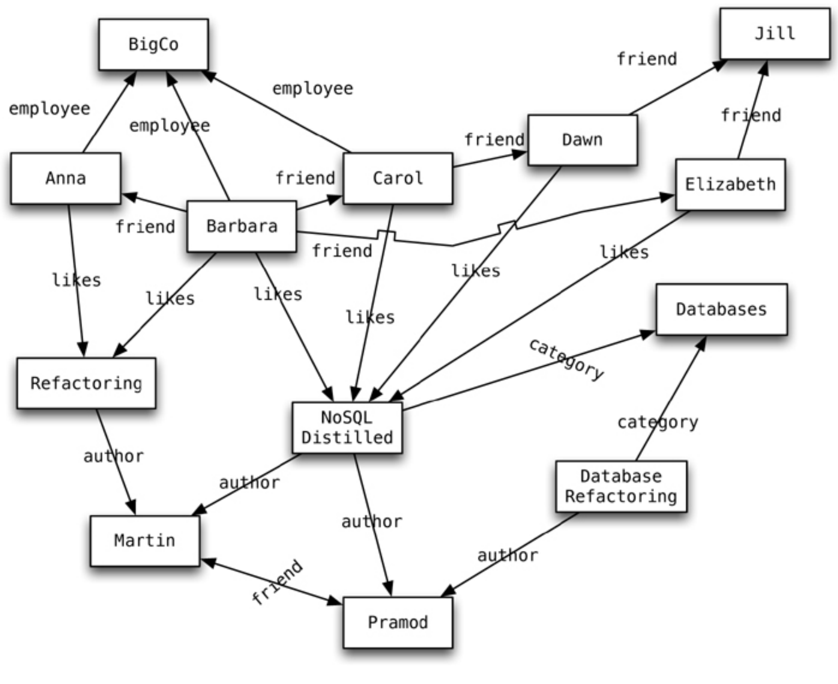


Figure 4.5: A graph structure example, symbolizing relationships in a social network [Sadalage and Fowler 2013].

4.3.3 Project Voldemort

In our peer-to-peer gaming solution, we require a distributed storage service that is fault-tolerant, highly available and scales elastically to the number of users. Below, we present Project Voldemort, a non-relational database management system we use to manage game state data in our peer-to-peer gaming network. Project Voldemort, commonly referred to as Voldemort, is a low latency key-value distributed system written in Java, that was developed by LinkedIn [LinkedIn 2012]. Tests conducted by LinkedIn show that on a single node Voldemort can read 19,384 requests per second and write 16,559 requests per second. Nodes in Voldemort are independent of each other, there is no coordination and no single point of failure, allowing for node failures to be transparent from the user. LinkedIn uses Voldemort for algorithms that derive insight from the social networks users. From this, LinkedIn is able to offer features that make the social network enjoyable and easier to use. These features include: *People You May Know*, where users are presented with set of other users they might know and would like to connect with, *collaborative filtering* showcases relationships between users for various recommendations such as collaborations.

Project Voldemort was inspired by Amazon's Dynamo distributed system [DeCandia *et al.* 2007]. Some of the schemes inspired from Dynamo include:

- Data replication: Data is automatically replicated over multiple nodes. A factor can be chosen as to the number of nodes each key-value data is replicated.
- Required reads: A set number of reads from nodes (performed in parallel) are required before a read is declared successful. This is used to resolve inconsistent data, discussed in detail below.
- Required writes: The least number of writes that can succeed without the client getting back an exception.
- Key-value serialization and compression: Voldemort can use different serialization structures for key and value data.
- Pluggable storage engines: Voldemort supports various read-write storage engines, including Berkeley DB (BDB) Java Edition and MySQL [Olson *et al.* 1999; Pachev 2007]. Voldemort also offers its own custom read-only storage, memory storage and cache storage.

Voldemort consists of a modular pluggable architecture, as shown in Figure 4.6. Each module is responsible for performing exactly one function. At the top of the architecture stack, there is the simple client API. The *conflict resolution* and *consistency mechanism* modules are used to deal with inconsistent data, discussed further in Section 4.3.3. The *routing* module handles the partitioning and replication of data, discussed below. The serialization module translates data so it can be persisted. Voldemort currently supports the following serialization formats: JavaScript Object Notation (JSON), strings, Java- serialization, Protobuf, Thrift, Avro and Identity. Voldemort also allows a programmer to code their own serializer for Voldemort. The final module is the persistent storage engine. Since each module is a separate entity, modules can be interchanged to meet different needs. For example, the routing module could be on the client side (creating a "smarter" client) or on the server side (traditional routing method).

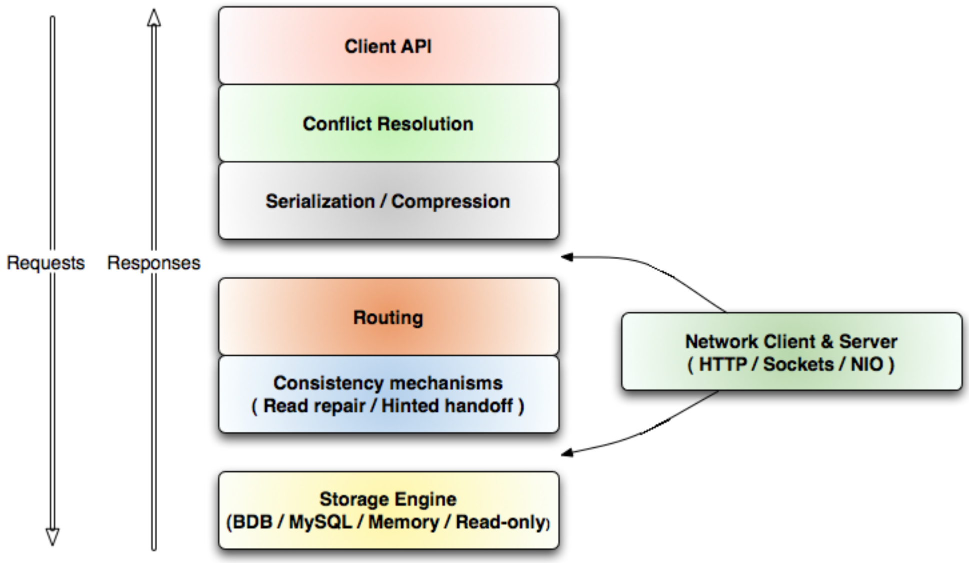


Figure 4.6: Volemort architecture containing modules for a single client and server [LinkedIn 2012].

Rabl *et al.* [2012] evaluated the performance of modern open-source storage systems, including: Apache Cassandra, HBase, Voldemort, Redis, VoltDB and MySQL. Rabl *et al.* [2012] observed linear scalability in Voldemort, Cassandra and HBase. Voldemort also exhibited the lowest latency out of the three systems. Cassandra had a higher latency but displayed the best throughput. HBase had the least throughput but showed low write latency in exchange for high read latency.

Partitioning and Replication

Voldemort partitions data across a cluster of nodes so that no single node holds the complete data set. If the data were to be put on a single disk, disk access for small values would be dominated by seek time. Partitioning improves cache efficiency by splitting the set of data into smaller chunks [LinkedIn 2012]. Servers in the Voldemort cluster are not interchangeable; therefore, requests need to be routed to a server that holds the requested data. Voldemort uses a consistent hashing scheme to partition data and allocate a partition of data to a node. The basic idea of consistent hashing is that data items and nodes are hashed into a logical ring [Kurose and Ross 2009]. The hashing algorithm determines where each data value has to be stored, see Figure 4.7. The same hashing algorithm is used to identify where the value is stored. If the value is not found on the calculated node, the client tries each node in the hash ring until it encounters the key-value tuple.

The consistent hashing algorithm works with a varying number of nodes. When a node fails or leaves the hash ring, the partitions are distributed equally (at best) amongst the remaining nodes. In the case of Voldemort, since data can be replicated amongst a number of nodes, the values of a failed node can be recovered. When a node is added to the cluster, partitions are split further and shared with the new node. Voldemort uses the MD5 hashing algorithm [Rivest 1992].

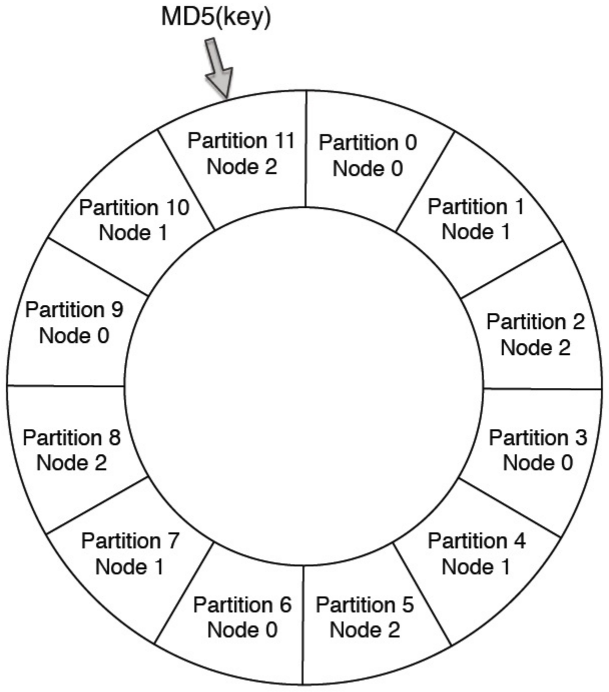


Figure 4.7: A Simple hash ring cluster topology for 3 nodes and 12 partitions [Sumbaly *et al.* 2012].

Voldemort has two forms of routing: server-side and client-side routing. Client-side routing retrieves metadata required for routing upon the client bootstrapping to the server. Fewer hops are then possible for retrieving values because the metadata allows the client to calculate the appropriate storage node.

Consistency and Versioning

Voldemort employs an eventual consistency data model. That is, Voldemort tolerates inconsistency in exchange for availability and partition tolerance. When a key-value entry is stored on Voldmemort, the tuple is replicated for availability and tolerance. Each value also contains a versioning vector clock [Fidge 1988]. Along with versioning, Voldemort uses the *read repair* conflict resolution process to reach consistency [Lakshman and Malik 2010]. The approach allows inconsistent values of data to be written on the Voldemort stores. When data is read, a query is made to several store nodes on the key to detect versioning conflict. If a conflict is detected, the values are updated to the latest value, thereby resolving data inconsistency. *Hinted handoff* is another technique used in Voldemort to reach consistency. During writes, if a destination node is down, the value is stored as a "hint" on a node that's alive. When the node which was down is restored, the "hint" is pushed onto the node to make the data consistent.

4.3.4 Summary

Large scale distributed systems require a solid persistent storage system. Essential requirements for a distributed storage system are scalability, availability and consistency. The CAP theorem provided an argument that only two of the three above properties can be satisfied in a distributed system. Consequently, distributed database systems have to trade-off one of the properties in favour for a weak or enforced consistency system. Non-relational database systems trade-off consistency for availability and scalability. Voldemort, a low latency, fault-tolerant, highly available, scalable key-value storage system contains many features which we seek in a distributed game state management system. But we also require that the game state be kept consistent.

Online multiplayer games face similar challenges to distributed data systems. They illustrate transactions on shared state, involving write traffic. Though this would cause an issue in a distributed game system where games states need to be highly available, there are two vital properties that are used to resolve the issue of availability. Firstly, games tolerate weak consistency in the application state. Current client-server game models provide a perfect example, where the implementations minimize interactive response time by presenting a weakly consistent view of the game world to players. With a weak consistency model, given a set period of time, the updates previously made to the data will propagate through the system. Secondly, game-play is usually governed by a strict set of rules that make the reads and writes of shared state highly predictable. For example, most reads and writes caused by a player occur on objects which are in the players AOI [Bharambe *et al.* 2006]. At any single point, players can only interact with a small portion of the game world. In most games, the notion of AOI focuses on a local view of the game opposed to the global view. Building on the concept of area of interest, we intend to provide a weak consistency globally in order to remain available. However, each area of interest will remain strongly consistent. Besides providing low latency, the weak consistency model also contributes to improving the overall performance of the system. The Supernode Control protocol that we base our system model on, favours strong consistency locally but weak consistency (eventual consistency) globally.

4.4 Models of Concurrency

In this section, we present concurrent programming paradigms that we can use to use to simultaneously handle a large number of network messages in our peer-to-peer gaming solution. In Section 4.4.1, we discuss the problem that gave rise to concurrent programming practices. Section 4.4.2 explores two forms of concurrent system paradigms, thread-based systems and event-based systems. We compare each paradigm and highlight why an event-based paradigm is suitable for our peer-to-peer gaming solution. We also briefly discuss research examining hybrid systems that combine thread-based and event-based paradigms. In Section 4.4.3 and 4.4.4 we present the event-based actor programming paradigm and introduce the Akka actor toolkit that we used to construct our peer-to-peer gaming solution. All the essential Akka features used in this research are discussed.

4.4.1 Overview

Concurrent computing is a computing paradigm in which programs are designed as collections of interacting computational processes that may be executed in parallel [Ben-Ari 1990]. An application that uses the concurrent programming model breaks up the system into tasks. All the subtasks are permitted to run in parallel with each other. Unlike sequential programming, the concurrent programming paradigm allows an application to take advantage of a multi-core processor and complete computations quicker. Within concurrent programs, concurrent components may require a form of communicating with each other in order to complete a task. These communication models are separated into two classes: shared memory and message passing communication. In the shared-memory programming model, processes communicate with each other by reading or writing data from a shared address space. In the messaging passing paradigm, processes exchange data by sending messages to each other.

4.4.2 Concurrent Programming Paradigms

There are traditionally two forms of concurrent computing systems: thread-based systems and event-driven systems [Erb 2012]. Thread based programming is characterized by protection and addressing mechanisms oriented towards procedure calls which take a process from one context to another [Lauer and Needham 1979]. Cooperation among processes is achieved by locks, semaphores or other synchronizing data structures. The event-driven programming model is a programming paradigm which consists of processes that react to events. Events are actions which can be observed by the process. These events could come from a number of sources, some of which are human input (for example clicking on a button), timers, observation upon shared state and receiving a message from a process. In an event-driven program, a central event loop watches all external sources and invokes callback functions to process each action as it occurs [Gustafsson 2005].

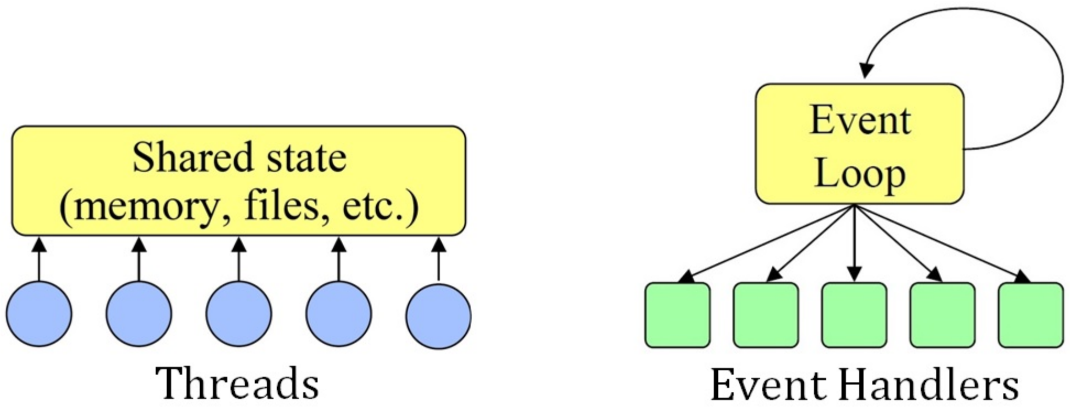


Figure 4.8: A comparison of thread and event based systems [Ousterhout 1996].

Threads versus Events

The debate between using threads or using events in a computing system is a very old one. In 1979, Lauer and Needham [1979] published an empirical analysis on message-oriented (event driven) systems and process-oriented (thread) systems. In the paper, it was shown that these two categories of computing systems are duals of each other and that a system which is constructed according to one model has a direct counterpart in the other. Lauer and Needham [1979] also state that both paradigms are logically equivalent (though they may be diverging concepts and syntax) and the performance of programs written in both models are identical (given that identical scheduling strategies are used). In other words, neither model is inherently preferable, and the main consideration for choosing between them is the nature of the machine architecture upon which the system is being built and the application which the system will ultimately support. Lauer and Needham [1979] defined a dual mapping of the two system models, these mappings are shown in Table 4.1. Despite neutrality the duality argument brought forth, it allowed many authors to further pinpoint problems in each model when dealing with specific applications.

Event-driven system	**Thread-oriented system**
Send message	Procedure call
Send reply	Return from procedure
Send message; await reply	Executing a blocking call
Waiting for messages	Waiting on condition variables (WAIT, SIGNAL)
Event handler	Monitor
Events accepted by a handler	Functions exported by a module
Event loop	Scheduler

Table 4.1: Lauer and Needham [1979] duality mapping of thread and event based systems adapted from von Behren *et al.* [2003] and Li and Zdancewic [2007] to resemble current event-driven systems.

The case for threads

Advocates of thread-based systems state that thread systems allow programmers to express control of flow and encapsulate state more naturally [von Behren *et al.* 2003]. A programmer can reason about the series of actions taken by a thread in the familiar way as with a sequential program. Event-driven programming, in contrast, is more difficult to reason with. Most general-purpose programming languages do not provide appropriate abstractions for programming with events [Li and Zdancewic 2007]. In order for a programmer to understand the flow of the system, an event-driven system is decomposed into multiple event handlers and represented as some form of state machine with explicit message passing or in continuation-passing style (CPS). Also, some event-driven systems *call* a method in another process by sending an event. A *return* from the process via a similar event mechanism is expected. These *call/return* pairs often require the programmer to manually save and restore the live state. This procedure, known as *stack ripping*, is a burden in programming languages that do not naturally support event-driven models [Adya *et al.* 2002].

A thread's run-time call stack encapsulates all live states for a task. This also allows a task's state to be easily cleared up after termination or debugging after an exception. Many programming languages support the development and debugging of threaded applications, which often drives developers away from event-driven applications. Threads are also well-known entities in operating systems, in which the exploit of multi-threading and parallelism prospered in order to run multiple tasks simultaneously [Silberschatz *et al.* 2008]. Most importantly, other concurrency methodologies rely on underlying thread-based implementations, although they hide this trait from the developer in order to utilize a multi-core CPU [Erb 2012; Typesafe 2013].

The case for events

Supporters of event-based systems state that event-driven systems provide better performance for high concurrency applications. Event-driven systems expose the scheduling of interleaved computations explicitly to the programmer, thereby allowing the programmer application-specific optimization that can improve performance [Li and Zdancewic 2007]. Thread-based systems often perform poorer for a number of reasons. The first case in threads poorer performance, when compared to event systems, is due to the high context switching overhead. The overhead is due to preemption, which requires saving registers and other states during context switches [Silberschatz *et al.* 2008]. Event handlers typically perform small amounts of work, so they require very little amounts of local storage. Compared to threaded systems, event-driven systems have minimal per-thread memory overhead and context switching costs.

Thread performance is also degraded by its shared state property. In order to avoid race conditions on shared data, thread systems employ locks. The locking overhead causes other threads to wait while a single thread operates on a specific set of data, in essence, blocking. With event systems, processes communicate by sending each other messages in order to share data. By not using blocking methods, event-driven systems allow full concurrency of processes. The use of locks and monitors in thread-based systems require highly cautious programming skills. Using locks and monitors raises the possibility of a system being deadlocked. In the context of using a concurrency model within a operating systems, event-driven systems are likely to be more portable. Threaded code might not be easily portable to another operating system due to different multi-threading libraries for different operating systems [Ousterhout 1996].

Li and Zdancewic [2007] state that event-driven systems are more flexible and customizable because the programmer has direct management to asynchronous operating system interfaces. Many high-performance I/O (such as Asynchronous I/O, e-poll and kernel event queues) interfaces provided by operating systems are asynchronous or event-driven [Li and Zdancewic 2007; Silberschatz *et al.* 2008]. With thread-based systems, the thread interface can be inflexible and the scheduler may also be hidden away from the programmer. Hiding the scheduler prevents a system from making optimal scheduling decisions based on a specific application. Through customization in event-driven systems, an application can schedule tasks, favour certain request and other optimization techniques. Though it was stated that thread stacks provide an easier way for debugging, threads stack are also an ineffective way to manage a live state [von Behren *et al.* 2003]. Thread systems usually face a trade-off between wasting virtual address space on large stacks and risking a stack overflow. Thread systems avoid this by using a dynamic stack growth or unwinding (removing entries) a stack. Another advantage in event-driven systems is that batch processing is possible by grouping similar events.

Summary

Thread-based systems naturally encapsulate state are widely used in existing systems [von Behren *et al.* 2003]. Multiple threads cooperate by accessing shared data. In order to prevent unanticipated system behaviour from threads manipulating shared data, thread based systems employ locks. The use of locks inadvertently decreases the performance of a system. Event-based systems do not use locks to share data. Consequently, event-based systems tend to provide higher performance compared to thread-based systems. Given that our network solution needs to process messages rapidly, we focus the rest of our research on event-based systems. In the next section, we introduce the event-driven actor programming paradigm.

Event-driven paradigms are able to provide higher concurrency performance compared to thread-driven paradigms. Event-driven paradigms, unlike thread-driven paradigms, do not require locks or monitors to share data. Event-driven models share data by passing messages between processes. By virtue of adopting the highly concurrent event-based paradigm, our system reduces the time taken to process network messages.

4.4.3 Actor model

The Actor model is a concurrent programming paradigm which uses autonomous and concurrent objects, called actors, to execute instructions asynchronously [Hewitt 2012]. In the actor model, actors are lightweight processes that encapsulate state and behaviour. Actors communicate exclusively by exchanging asynchronous messages (some actor models allow exchanging of synchronous messages). Messages are buffered in a mailbox, from which an actor retrieves the message for processing. A message can change the state and behaviour of an actor. Figure 4.9 shows the common properties of an actor.

Actor objects hold instance variables which can reflect the state of an actor. The state can be characterized by a counter, set of listeners, pending requests, an explicit state machine and so forth. Since actors are isolated from other processes, they do not have shared state. Because the internal state is vital to an actor's operations, having inconsistent state is fatal. Thus, when the actor fails and is restarted, the state will be created from scratch, like upon first creating the actor [Typesafe 2013]. This signifies a self-healing system. An actor's behaviour defines the action to be taken when processing a message.

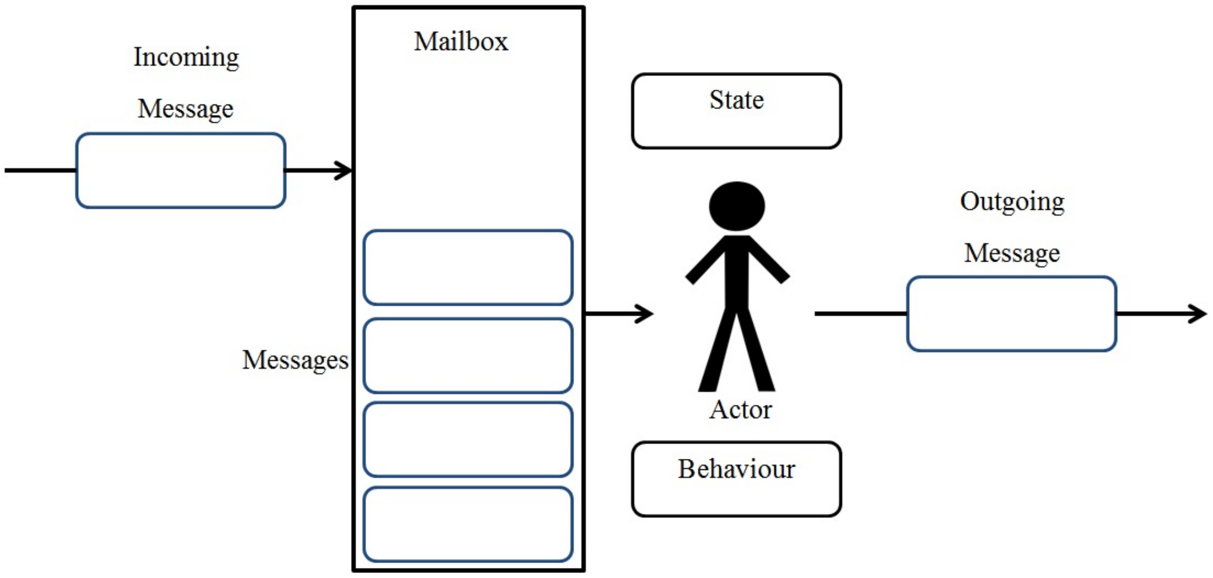


Figure 4.9: Illustration of the actor model, adapted from Ridgway [2011].

The behaviour of an actor may also change due to the reaction from a specific message. This implies that a message can change an actor's behaviour for a new behaviour. An actor defaults to the original behaviour when restarted. Actors provide fault-tolerance in the sense that if one actor crashes the rest of the actors continue to operate. Actors are also inherently concurrent, this allows actor systems to scale.

The actor model is not a new concept. In 1973, Hewitt *et al.* [1973] published a paper introducing the concept of actors. Clinger [1981] states the development of the actor model was not only motivated by the message-passing model of concurrent computation, but also the prospect of highly parallel computing machines consisting of dozens, hundreds or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network. The actor model became the premise of programming languages that required high level concurrency such as Erlang [Armstrong 2010]. Later on, Erlang was used in the AXD 301 telecom switch which achieved a reliability of 99.9999999 percent (nine nines) [Armstrong 2007].

4.4.4 Akka

Akka is an actor based toolkit for building highly concurrent, distributed, and fault tolerant event-driven applications on the Java Virtual Machine (JVM) [Venners 2011]. Akka is designed to be used on systems that are highly transactional, require high throughput and low latency. Akka was created by Jonas Bonér, who was inspired by Erlang's highly concurrent and event-driven applications [Bonér 2011]. Akka is currently part of the Typesafe platform along with Scala and the Play Framework. Through experiments conducted by Nordwall [2013a], Akka actors have exhibited a high transaction rate. The tests showed that an Akka system can process more than 50 million messages per second (through configuration). Also, Akka actors use a small amount memory, with an approximated 2.5 million Akka actors consuming a single GB of heap memory.

Products/businesses which make use of Akka include Gilt (online shopping), The Guardian (news media), 47 Degrees (mobile application development), Heluna (anti-spam email service) and many more. Akka has been deployed in various systems for the following uses:

- Transaction processing: online gaming, finance/banking, trading, statistics, online betting, social media and telecoms.
- Service backend: REST services, SOAP, CometD, WebSockets and so forth.
- Concurrency/parallelism.

- Simulation: Grid computing, MapReduce, master/slave etc.
- Batch processing: Apache Camel integration to hook up with batch data sources [Anstey 2009].
- Communication hubs: Telecoms, web sites, mobile media.
- Online game servers.
- Business intelligence and data mining.

Behind the scenes, Akka runs sets of actors on sets of real threads. Typically many actors share one thread, and subsequent invocations of one actor may end up being processed on different threads. Akka ensures that this implementation detail does not affect the single-threadedness of handling an actor's state [Typesafe 2013]. Akka implements actors as a reactive, event-driven, lightweight thread that shields and protects the actor's state. Akka actors provide the concurrent access to the state allowing programmers to write code without worrying about concurrency and locking issues [Gupta 2012].

The Akka framework contains a number of features. For example, Akka offers transaction support where Akka implements transactors that combine the actors and software transactional memory (STM) into transactional actors. More features continue to be added with the rapid releases of newer Akka versions. The following sections will only discuss Akka features and concepts which are related to our peer-to-peer gaming solution.

Akka Hierarchy

The Akka actor system naturally forms a hierarchical actor model. An actor, which has a certain function in a program, might want to split up its task into smaller more manageable pieces. For this purpose it starts child actors which it supervises that handle the smaller tasks. These child actors may also have tasks which can be further divided to its own child actors and so forth. This essential feature of the Akka actor system allows tasks to be split up and delegated until they become small enough to be handled in one piece [Typesafe 2013]. Not only does the Akka hierarchical model allow tasks to be clearly structured, it also allows the resulting actors to be reasoned about in terms of which messages they should process, how they should react normally and how failure should be handled. If one actor does not have the means for dealing with a certain situation, it sends a corresponding failure message to its supervisor, asking for help. The recursive structure allows failures to be handled at the right level. Section 4.4.4 discusses supervision methods during an actor's failure in detail.

Messaging Model

An actor reacts to a message which it has received through a mailbox. Akka consists of four primary message types: *tell*, *ask*, *kill* and *poison pill*.

1. **Fire and Forget - *Tell***

   A one-way message, the producer of the message sends a message to the consumer. The producer does not expect a reply from the consumer. If the consumer wishes to respond to the producer, similarly it will send a *tell* message to the producer. Figure 4.10 illustrates a *tell* message between two actors.

2. **Send and Receive - *Ask***

   A two-way message, the producer of a message expects a reply from the consumer. The *ask* message naturally blocks a producer actor while it waits for a response from the consumer actor. Blocking is discouraged as it causes performance issues. The Scala *future* library is used to solve

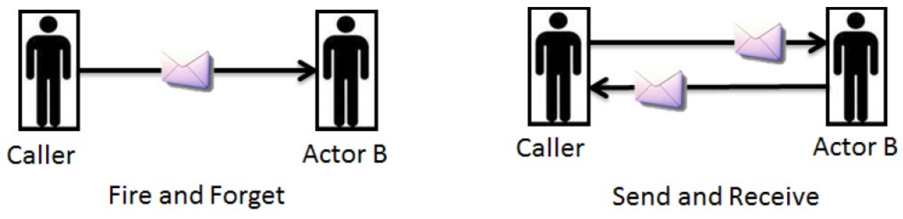


Figure 4.10: Diagram exhibiting a *tell* message (left) and a *ask* message (right) [Gupta 2012].

this issue. A *future* is an abstraction which represents a value which may become available at some point. A *future* object either holds a result of a computation or an exception in the case that the computation failed [Haller *et al.* 2013]. In Akka, a *future* is used to retrieve the result of an asynchronous operation. The result from an *ask* request can then be accessed synchronously (blocking) or asynchronously (non-blocking). Figure 4.10 illustrates an *ask* message between two actors.

3. ***Kill***

   A *kill* message is a synchronous message. The message causes an actor to kill itself and throw an exception, which can be handled by the actor's supervisor.

4. ***Poison Pill***

   A *poison pill* is an asynchronous message which initiates an actor's shut down. The poison pill does not allow an actor to be restarted after it has been shut down.

Actor Lifecycle

Akka actors go through a set lifecycle, which is summarized as follows: An actor is created and started with the *actorOf()* call. An actor's incarnation is identified by a path reserved by the programmer and a random Unique Identifier (UID). An actor can be restarted a number of times, replacing the actor's current state with the actor's initial state. When an actor is restarted, the content of the actor's mailbox is unaffected. The lifecycle of an actor ends when the actor is stopped. Figure 4.11 displays a descriptive diagram of an actor's lifecycle.

Akka provides several hooks which can be configured to change an actor's behaviour/state during certain phases of its lifecycle:

- Start Hook: When an actor is started the *preStart()* method is called, allowing the initialization of logic.
- Restart Hooks: When an actor restarts, the *preRestart()* method is called, including a message with the cause of error if the restart was caused by an exception. The *preRestart()* method can be used to notify services, prepare to hand-over data to the fresh actor instance [Typesafe 2013]. By default, the method stops all the actor's children and then calls *postStop()*. After the actor has been initialized with a new actor instance, the *postRestart()* method is called. By default, the *postRestart()* method invokes the *prestart()* call just as in the normal start-up case.
- Stop Hook: After stopping an actor the *postStop()* method is called. The method may be used, for example, deregistering the actor from other services; thus allowing for the clean-up of resources.

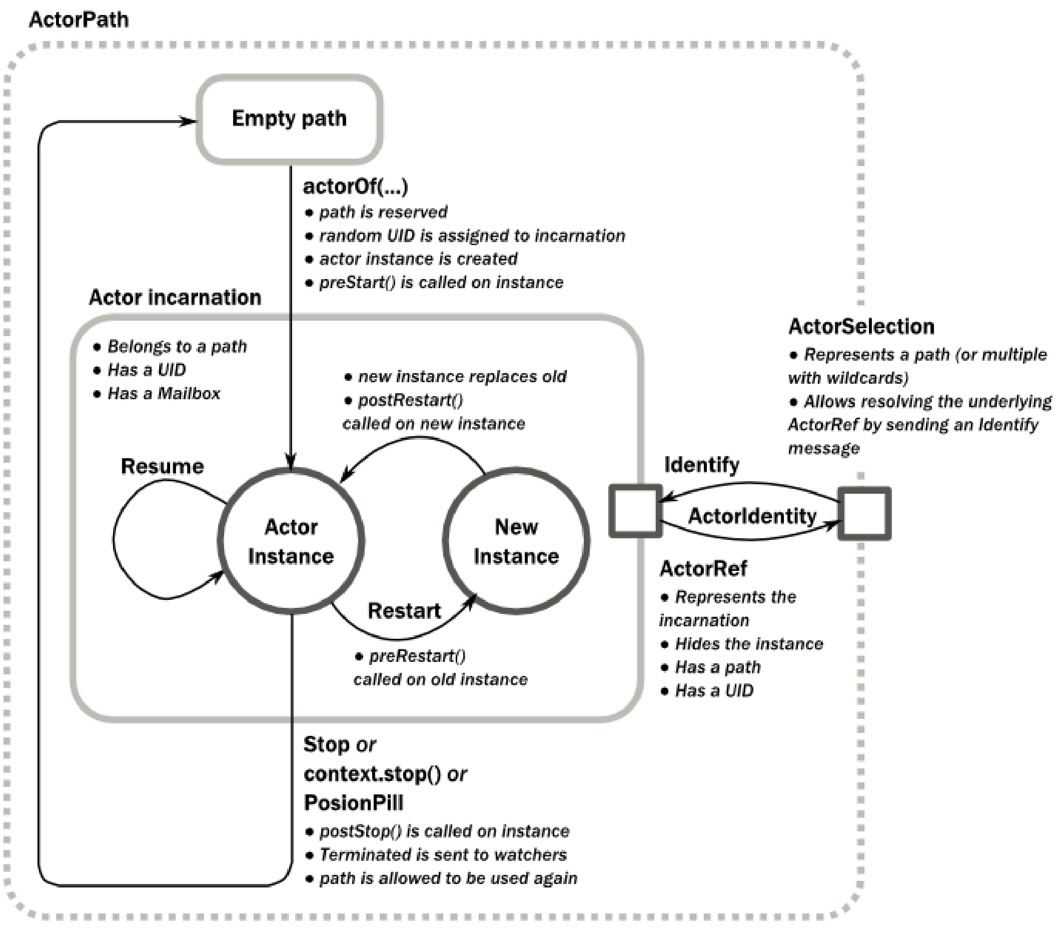

ActorPath
Empty path
actorOf(...)
• path is reserved
• random UID is assigned to incarnation
• actor instance is created
• preStart() is called on instance
Actor incarnation
• Belongs to a path
• Has a UID
• Has a Mailbox
• new instance replaces old
• postRestart() called on new instance
Resume
Actor Instance
New Instance
Restart
• preRestart() called on old instance
ActorSelection
• Represents a path (or multiple with wildcards)
• Allows resolving the underlying ActorRef by sending an Identify message
Identify
ActorIdentity
ActorRef
• Represents the incarnation
• Hides the instance
• Has a path
• Has a UID
Stop or
context.stop() or
PosionPill
• postStop() is called on instance
• Terminated is sent to watchers
• path is allowed to be used again

Figure 4.11: Akka actor state diagram [Typesafe 2013].

Fault-tolerance

Akka adopts Erlang's actor supervision model and "let it crash" philosophy. Distributed applications are bound to fail, not just from software errors but hardware failure as well. Instead of using a defensive programming model, Akka allows actors to crash. By using supervisors and processes that monitor actors, Akka is able to restart failed actors and return the system back into normal operation.

Supervision

As discussed in Section 4.4.4, Akka has a hierarchical relationship between actors. A parent actor may break down a large task and delegate the sub-tasks to the child actors. The parent actor supervises and manages the lifecycle of its subordinates. Therefore the parent actor must respond to the failure of its subordinates. When a subordinate actor fails (for example, throws an exception), it suspends itself (and all its subordinate actors) and sends a message to its supervisor signalling failure. When a supervisor is informed of a subordinate's failure, depending on the nature of the failure, the supervisor can take the following actions [Typesafe 2013]:

- Resume the subordinate actor, keeping the actors state and resume operation as if nothing has happened.
- Restart the subordinate actor, clearing out its accumulated internal state.
- Terminate the subordinate actor permanently.

- Escalate the failure to its own supervisor, thereby failing itself.

The actions above represent supervision strategy actions. Supervision strategies define how the failure of the child actors are handled, how often the child is allowed to fail, and how long to wait before the child actor is recreated [Gupta 2012].

Akka consists of two classes of supervision strategy: One-For-One strategy and All-For-One strategy. The One-For-One strategy implies that in the case of any one actor under a supervisor fails, the failure strategy is applied to that actor only. The All-For-One strategy implies that in the case of any one actor under a supervisor fails, the failure strategy is applied to all the actors under the supervision. Figure 4.12 provides an example between the two Akka strategy classes. Actor A uses a One-For-One supervision strategy, whereas actor B uses an All-For-One supervision strategy. Actors A2 and B2 both encounter a failure. With subordinate actors of A, only the actor A2 is dealt with in resolving the fault. So actors A1 and A3 continue to operate normally. In this case of the actor B's subordinate actors, the fault solution is performed on all the subordinate actors of B.

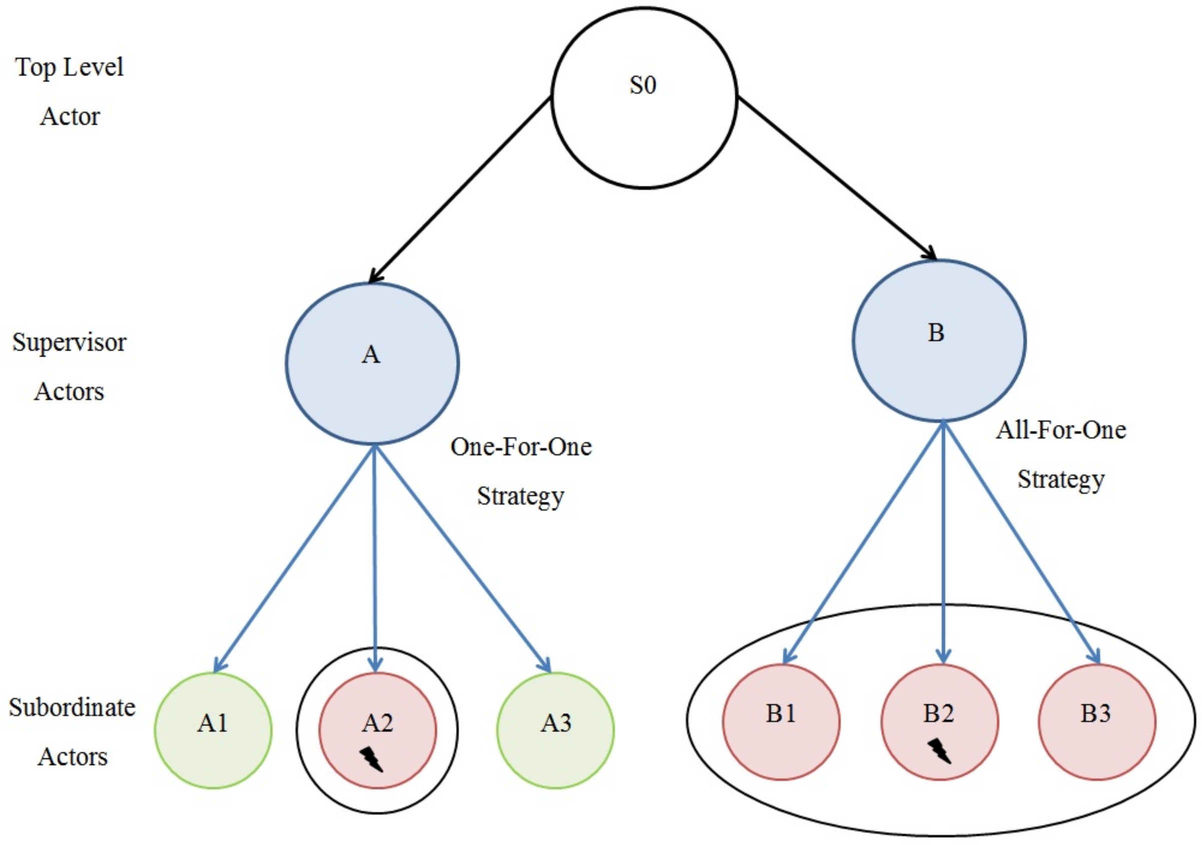


Figure 4.12: The difference between the One-For-One strategy and All-For-One strategy. Adapted from Gupta [2012]

The All-For-One strategy is applicable in cases where the ensembles of children have a tight dependency among them, that a failure of one child affects the function of the others. For example, in a transaction application where data is processed in a series of steps, a failure on one of the steps leads to inconsistency of state in the other actors. In this case, it is appropriate to restart all the actors to make sure that they all have a consistent state.

Lifecycle Monitoring

The monitoring strategy provides a mechanism in which an actor can listen for certain events from another actor. In contrast to the special relationship between a parent and a child actor, in terms of supervision, any actor may monitor another actor. Since actors emerge from creation fully alive and restarts are not visible outside of the affected supervisors, the only state change available for monitoring is the transition from alive to dead [Typesafe 2013]. Monitoring is thus used to tie one actor to another so that it may react to the other actor's termination. Examples of instances in which actor monitoring should be used are:

- When a supervisor cannot simply restart a child actor and has to terminate it, for example when errors occur during actor initialization. In this case, the supervisor should monitor the child actor and re-create it or schedule itself to retry at a later time.
- When an actor stops because of an external event such as a poison pill.
- When an actor is not part of the hierarchy but wishes to be notified of an actor's state.

Location transparency

Akka actors are location transparent in respect to their placement across multiple Java virtual machines and network nodes. Akka actors allow an application to scale up and use all the processing power underlying a machine's hardware. Once an Akka application reaches a machine's hardware limit, Akka actors can be distributed among multiple machines, further increasing the scalability on an Akka application. This is referred to as remoting.

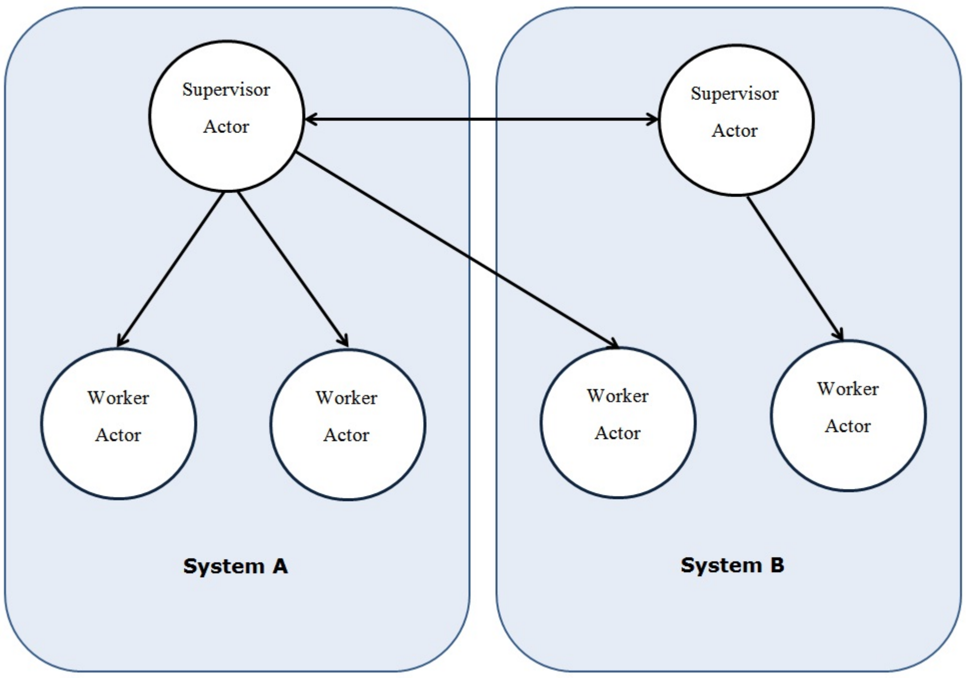


Figure 4.13: Illustration of Akka actors location transparency property.

Akka persistence

A recent, and experimental, feature added to the Akka library is *Akka persistence*. The *Akka persistence* feature enables stateful actors to persist their internal state so that the state can be recovered when an actor is started or restarted by a supervisor [Typesafe 2013]. For example, actors can recover their state after the JVM crashes. Messages which actors receive are logged in a journal. by replaying these messages, actors are able to rebuild their internal state. Snapshots of an actor's state can also be used to recover the actor's internal state. The key concept behind *Akka persistence* is that only changes to an actor's internal state are persisted but never its current state directly (except for optional snapshots). These changes are appended to storage, nothing is ever mutated, which allows for very high transaction rates and efficient replication.

The *Akka persistence* also ensures message delivery to an actor after crashes. If an actor never received a message which was logged in the journal before the crash, the persistence library can certify that it will deliver the message. The *Akka persistence* feature is inspired by the Eventsourced library [Krasser 2013]. The *Akka persistence* library follows the same concepts and architecture of Eventsourced but significantly differs on API and implementation level, making actor persistence easier to implement in Akka using the *Akka persistence* library.

Mailboxes

An Akka mailbox holds the messages that are destined for an actor [Typesafe 2013]. When an actor sends a message to another actor, the message gets enqueued into the receiving actor's mailbox. The receiving actor proceeds to dequeue the message for processing. Normally each Akka actor has its own mailbox, but there are Akka features which allow multiple actors to share a single mailbox. Akka offers several mailboxes implementations, each targeting specific Akka user's needs. The different mailbox implementations are separated into two categories, bounded and unbounded. A bounded mailbox limits the number of messages that can be queued in the mailbox, meaning it has a defined or fixed capacity for holding the messages [Gupta 2012]. If an actor sends a message to a bounded mailbox that is full, the sending actor would be blocked from sending messages to the receiving actor until the receiving actors mailbox indicates that there is available space in the mailbox for the message to be received.

The Akka default is to dequeue messages from the mailbox in the order the messages were received. For applications that require specific messages to be prioritized above other messages, a priority mailbox may be used. The priority mailbox dequeue's messages based on the priority assigned to the message. Akka also offers a durable mailbox which stores mailbox messages in a persistence storage. Given an actor which uses a durable mailbox, if the system were to crash and then restarted, the actor would be able to continue processing the pending messages in the mailbox as if nothing had happened. If the given Akka mailboxes implementations do not satisfy the needs for an application, one could create their own mailbox implementation.

Dispatchers

Dispatchers are defined as communication coordinators that are responsible for receiving and transmitting reliable messages. For example, an air traffic controller coordinates the use of a runway between the various airplanes landing and taking off. In Akka, a dispatcher controls and coordinates the messages sent to the actors mapped to underlying threads [Gupta 2012]. Dispatchers in Akka are based on the Java Executor framework, which provides a structure for the execution of simultaneous tasks. Akka dispatchers ensure the resources are optimized and messages are processed as fast as possible. The dispatchers run on threads, in which they dispatch actors and messages from the attached mailbox and allocate on heap to the executor threads [Gupta 2012]. Figure 4.14 shows the relationship between dispatchers, actors, mailbox and threads. The executor threads are configured and tuned to the underlying processor cores that are available for processing the messages.

Akka provides four types of dispatchers, which can be customized [Typesafe 2013]:

- Dispatcher: The default dispatcher, an event-based dispatcher that binds a set of Actors to a thread pool.
- Pinned dispatcher: The dispatcher dedicates a unique thread for each actor using it; that is, each actor will have its own thread pool with only one thread in the pool.
- Balancing dispatcher: This is an executor-based event-driven dispatcher that will try to redistribute work from busy actors to idle actors.

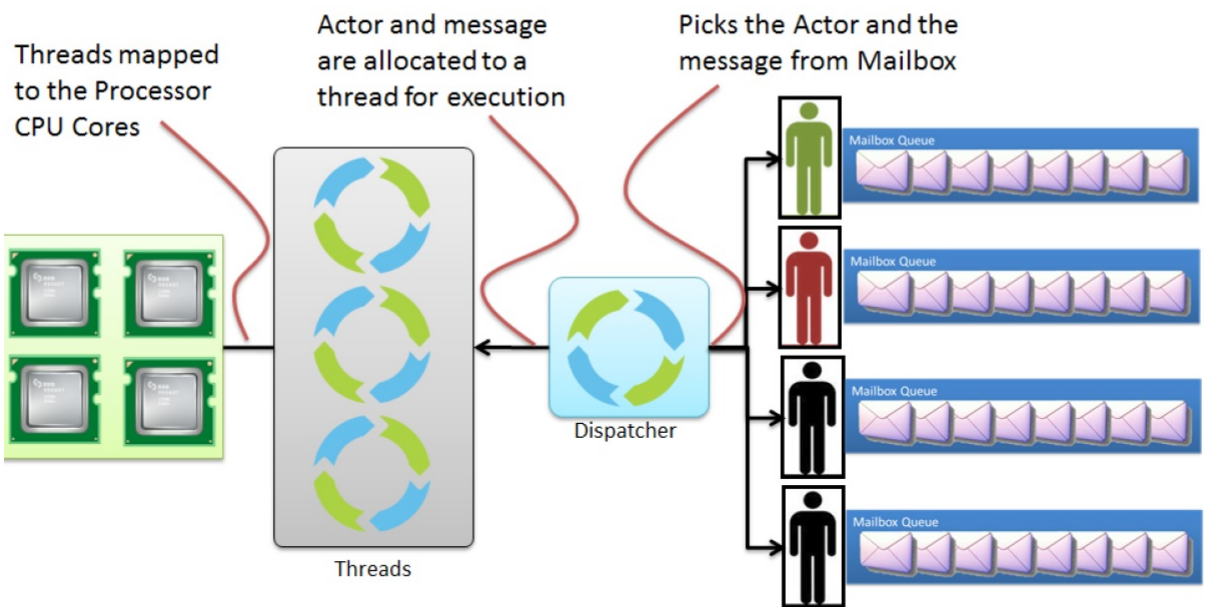

Threads mapped to the Processor CPU Cores
Actor and message are allocated to a thread for execution
Picks the Actor and the message from Mailbox
Mailbox Queue
Mailbox Queue
Mailbox Queue
Mailbox Queue
Dispatcher
Threads

Figure 4.14: Graphical explanation on the role of a dispatcher in Akka [Gupta 2012].

- Calling thread dispatcher: The dispatcher runs invocations on the current thread only. This dispatcher does not create any new threads, but it can be used from different threads concurrently for the same actor.

Routers

A router is a type of Akka actor that receives messages and routes the incoming messages to other actors, known as the routers' routees [Typesafe 2013]. An Akka router can create several instances of the same actor class. This enables an application to spread the work load across multiple actors, see Figure 4.15. Given a router with a pool of routees, there are several routing strategies which can be used by the Akka router. Akka offers the following message routing strategies:

- Round Robin router: Messages are routed to each routee in a circular order [Silberschatz *et al.* 2008].
- Random router: The router randomly selects a routee and forwards the received message to the routee.
- Smallest Mailbox router: The router identifies and sends a message to the routee with the least the number of messages in its mailbox.
- Broadcast router: Messages are sent to *all* the routers routees.
- Scatter Gather First Completed router: The router sends the message to *all* its routees as a *future*. It then waits for a routee to respond. Whichever routee responds first, the result from that routee will be used.
- Consistent Hashing router: The router uses consistent hashing to select a routee based on the received message [White 2007].

Routers that create multiple instances of routees can have the number of routees fixed or have the number of routees change dynamically dependent on the load of messages. Routees provide a resize strategy in which the number of routees can be increased or decreased on a set upper and lower bound. Resizing is triggered by sending messages to the actor pool, but it is not completed synchronously; instead a message

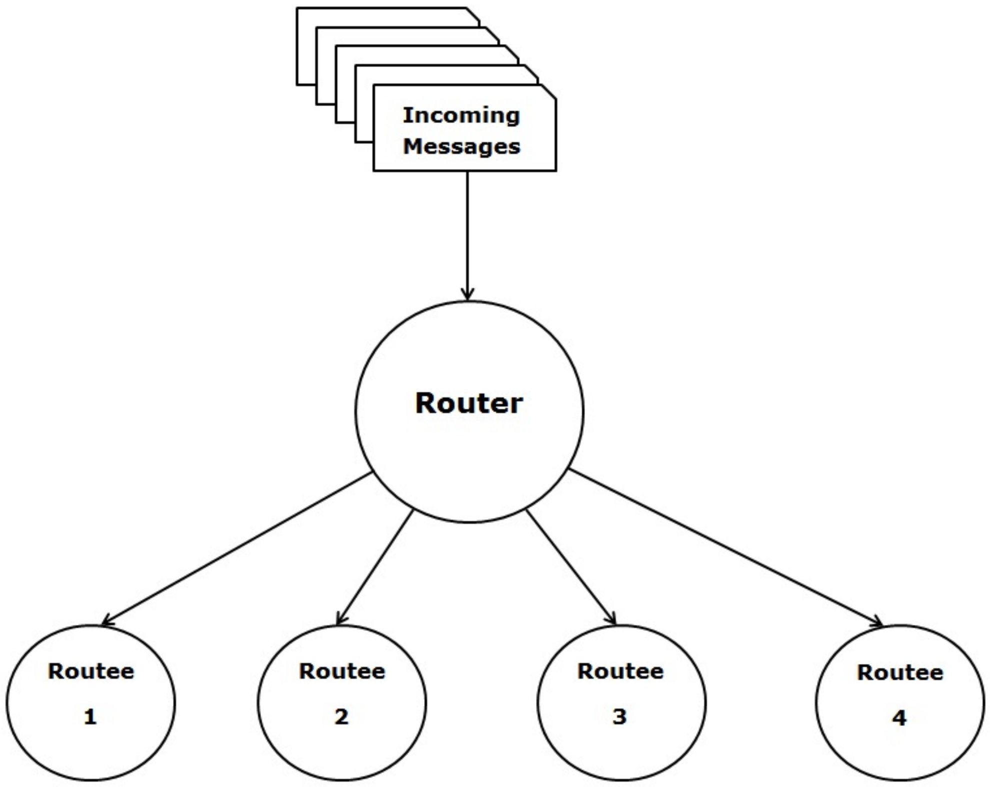


Figure 4.15: A simple outline of Akka routers. Messages are received by the router and sent to a routee dependent on the routing algorithm used. Adapted from Gupta [2012]

is sent to the “head” router to perform the size change [Typesafe 2013]. Consequently, the resizing of routees does not happen instantaneously because the resize message is queued to the mailbox of a busy actor. To remedy this, a balancing dispatcher could be used.

4.4.5 Summary

The challenge in processing a large number of messages received from several hosts is characterized by the concurrency model of the receiving host. Our peer-to-peer solution needs to process and respond to a large amount of messages without delaying the game play. Therefore, we examined different concurrent programming paradigms. The event-based paradigm allows for highly available concurrent operations and provides simplicity in developing applications without the use of locks; therefore, we focused on the event-based paradigm as the message processing model for our peer-to-peer gaming solution. We expressed interest in the actor the actor programming model that sends messages between processes called actors to complete a task. In particular, we examined the Akka actor toolkit that is designed for systems that are highly transactional, require high throughput and low latency.

The Akka toolkit is used for the implementation of the the proposed peer-to-peer gaming infrastructure given that the services offered by Akka (such as scaling and fault-tolerance) meet the requirements that we seek. In addition, the Scala programming language (a functional language, which naturally suits the actor model) is used to implement the system. There are several reasons why we choose to implement the network application using Akka, the first being the event-based actor model Akka employs. Since event-based systems pass messages and do not require locks to share data, an application is likely to be highly available. The message orientated system displays attributes that can be mirrored to the Internet Protocol suite. A message is passed on from actor to actor where the message is processed; similarly, packets are passed on from one layer to another where a specific process takes place. Akka allows us to create multiple instances of an actor that performs a specific function, and thereby distribute the workload which in turn decreases the amount of time taken to complete a task. Another benefit to using Akka (and Scala) is that it is independent from operating systems as it runs on the JVM. This means the distributed network application can be easily ported to another system as long as there is Java support. Scala also allowed easy integration of Voldemort, as Voldemort also runs on the JVM.

4.5 Conclusion

This chapter provided an overview of the key concepts that form the basis of our proposed peer-to-peer gaming network. We reviewed distributed database models and concurrency programming models. We also presented protocols designed to reduce resource consumption (WebSocket) and decrease the time taken to transfer web content over TCP (SPDY). The next chapter provides a walk-through on the design of our peer-to-peer gaming solution.

Chapter 5

System Design and Architecture

5.1 Introduction

In this chapter, we extend the abstract design of our peer-to-peer solution presented in Chapter 3 and discuss the components of our peer-to-peer solution in detail using the building blocks from the previous chapter. First, in Section 5.2 we give an overview of the components in the architecture and relate them back to the research aims. We also discuss the overall setup of Akka attributes that affect the performance and reliability of the system. In Section 5.3, we introduce the front-end component designated with directly interacting with messages sent to the system and redirecting each message to the appropriate system service. In Sections 5.4, 5.5 and 5.6, respectively we discuss the persistent storage, message dissemination, and service routing functions of our peer-to-peer solution. The actors in each function are also discussed. The design of the supernode setup is presented in Section 5.7. Finally, Section 5.8 details how the peer-to-peer network application can be obtained and executed by the reader.

5.2 System Overview

As discussed in Section 3.4 of Chapter 3, three key services were identified which each node in the peer-to-peer solution offers. The services offered are a distributed key-value storage system, a messaging service and a message routing service. The storage service module within the system consists of actors that are involved in the storage and retrieval of key-value data from a Voldemort storage database. The messaging module contains actors that deal with the publish-subscribe protocol, which allows a host to broadcast a message to several hosts. The actors contained in the service routing module provide a service in which a node routes messages to a service outside the peer-to-peer gaming solution. In Chapter 3, we also noted that our peer-to-peer solution is based on the Supernode Control protocol. Figure 5.1 extends the abstract design from Chapter 3 and introduces the building blocks discussed in the previous chapter into our peer-to-peer solution design. A Voldemort storage node may be placed in several locations dependent on the requirements of the application and available hardware in the peer-to-peer system. For example, the Voldemort storage node may be placed on the same machine as a supernode, or on any other peer involved in the peer-to-peer gaming network. A peer-to-peer infrastructure may also choose to have several machines dedicated to hosting the Voldemort storage, which can be used to store information such as a player's login details and a player's total score. Figure 5.1 also shows the interaction between the players and a supernode's functions.

Given the services mentioned above, a front-end service is required for a node to direct any incoming message to the correct service. Therefore, a fourth service, labelled as the System-Coordinator, is responsible for directing messages to the correct service within a node. Figure 5.2 gives a high-level overview of a node in the peer-to-peer gaming solution. More intricate details of the services are discussed further in this chapter. The API used in the peer-to-peer gaming solution can be found in Appendix A.

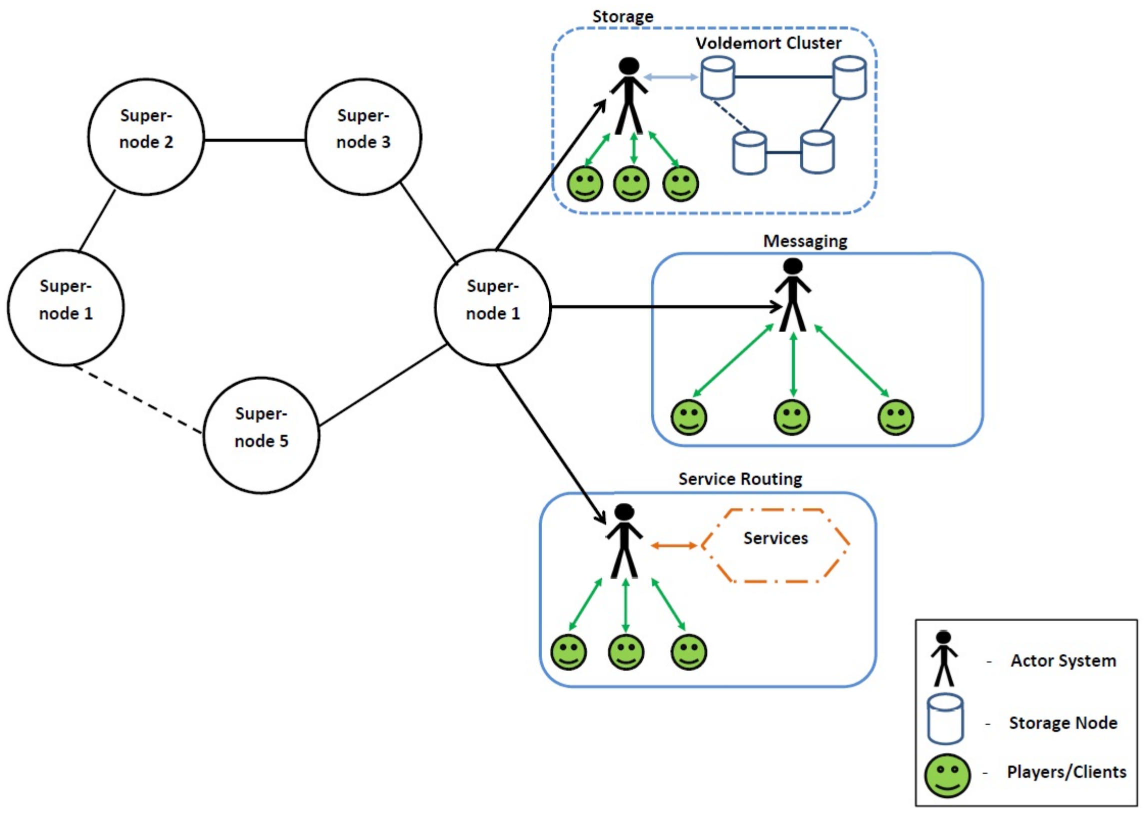


Figure 5.1: Overview of the peer-to-peer gaming system.

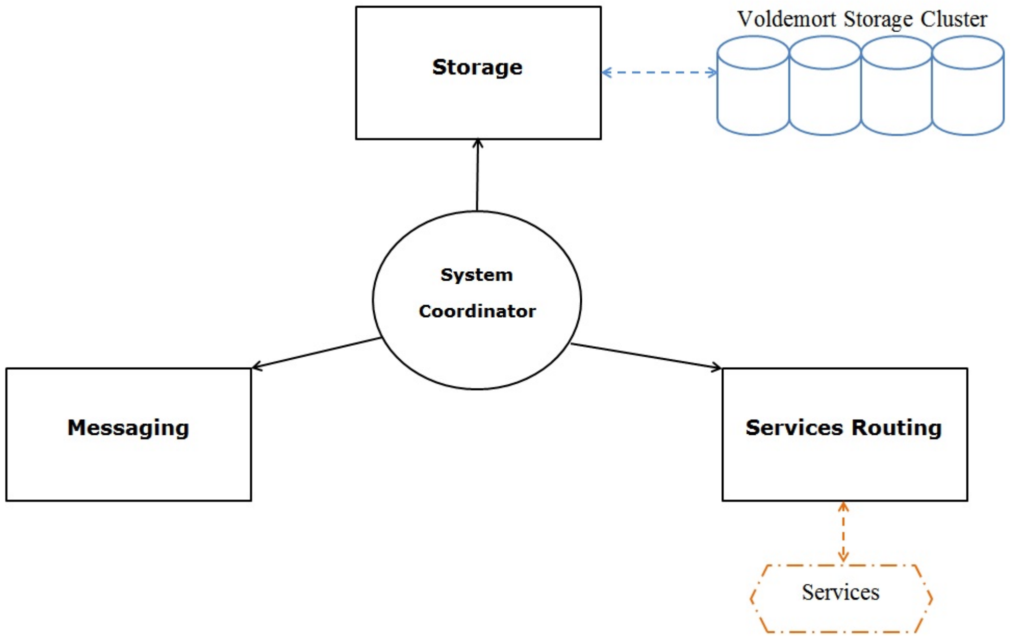


Figure 5.2: Basic overview of a node in the peer-to-peer gaming solution.

5.2.1 Communication Protocol between Nodes

In Chapter 3, it was stated that the SPDY protocol would be used to reduce the latency of reliably delivering messages from one node to another. Within SPDY, the WebSocket protocol was also used. Besides providing bi-directional communication between nodes in the Application Layer, the WebSocket protocol is required for the messaging service. The justification is discussed in Section 5.5.

Voldemort Limitations:

Voldemort services can use two types of network sockets for communication: a Java Non-blocking Input/Output (NIO) TCP socket and an HTTP socket. Unfortunately during the research, Voldemort's HTTP socket service was still under development. Due to this, Jetty's proxy service for SPDY could not be used. An attempt was made to replace Voldemort's TCP socket service with Jetty's SPDY application service, but due to time constraints, this task was abandoned. Voldemort's NIO non-blocking TCP was used for data communication with the Voldemort storage nodes.

5.2.2 Akka dispatcher

The default Akka dispatcher was employed in the network application, backed by the Java Fork/Join executor, which gives the desired performance in most Akka based applications [Typesafe 2013]. The default dispatcher assigns each actor with its own mailbox. If necessary, the dispatcher type and its attributes can be reconfigured through the Akka configuration file, discussed further in Section 5.2.6.

5.2.3 Routers

Actors which are created as routers have several instances of child actors (routees) that perform specific tasks. Initially, the Smallest Mailbox strategy was chosen to route messages to actors. The logic behind choosing the Smallest Mailbox strategy was that a message would be sent to a routee which had a few amount of items left to process. In doing so, a message had the chance of being processed in a much shorter time than it would with the Round-Robin strategy. During preliminary tests, it was noted that as the number of routees per router increases, the latency of the message response time actually grew. It was deduced that the Smallest Mailbox strategy spends some time inspecting which routee has the smallest inbox. The Smallest Mailbox strategy is likely more efficient when the time taken for a routee to process a message is larger than the time it takes for the router to compute which routee has the smallest mailbox. In our case, many of the actors do small and quick jobs, so the Smallest Mailbox strategy was not suited for the application. Thus the Round-Robin strategy was used because the router strategy knows (without any form of calculation) which routee is to receive the next message.

In the remaining sections of this chapter, actors that have several instances (a cardinality greater than one) are indicated with a $(*)$ in the actor diagram models. Actors with a single instance (a cardinality of one) are indicated with a (1).

5.2.4 Mailbox

Given that the system or host machine may fail, using a durable mailbox across the system would increase the level of fault-tolerance. The durable mailbox stores mailbox items on a durable medium and allows an actor to continue processing pending messages after a systems failure. However, it was discovered that in exchange for increased message delivery guarantee, durable mailboxes disproportionately decrease the time it takes to process a message (estimated to be 10 times slower) [Nordwall 2013b]. Given that the main goal of this research is to deliver messages in a timely manner, durable mailboxes were not used. The default Akka mailboxes, unbounded mailbox, were used across the whole system.

5.2.5 Fault-Tolerance

The actor fault-tolerance technique employed in the peer-to-peer gaming solution is taken directly from the Akka supervision model, as discussed in Section 4.4.4 of Chapter 4. Each child actor in the peer-to-peer solution is supervised by a parent actor. If an actor were to fail, the supervisor is to handle the failure. The supervision strategy employed across the peer-to-peer gaming solution is as follows: all the supervisors use a one-to-one fault solution strategy. If a child actor were to fail, the failure solution is performed on that specific child actor only. This prevents an actor failure affecting the sibling actors as well. When an actor encounters a failure, the default action is to restart the actor. Certain actors require further attention, and not just an actor restart, when an actor fails due to specific reason. The strategy taken with these actors is discussed further in the chapter.

In order to prevent an actor repeatedly failing, a limit is put on the number of times an actor may fail in a given time frame. In the peer-to-peer gaming solution an actor is allowed to fail ten times under ten seconds. If an actor were to fail more than ten times within the ten second time limit, the whole actor system on a node is shut down. An actor repeatedly failing and being restarted shows a form of logical error. Therefore, in order to prevent an actor being stuck in a restart loop, the system is shutdown so that the user can troubleshoot the issue.

Apart from the Akka actor supervision, another form of fault-tolerance was sought to provide fault-tolerance from sources outside the Akka system, such as the JVM crashing or machine failure. As stated in Section 5.2.4, the durable mailbox was intended to be used on the system to provide message delivery guarantee in the case of failure. However, it was also noted that the durable mailboxes decreases the speed of processing messages, therefore unbounded mailboxes were used. *Akka persistence* can be used to provide message delivery guarantee and actor state persistence.

At the time of this research, the *Akka persistence* library was in an experimental phase. Though it could be used, it may contain bugs and performance issues. When the experimental *Akka persistence* library was integrated into our existing system, the system began displaying errors which we could not resolve. Due to this, the *Akka persistence* was abandoned but is planned on being integrated once the final release of the library is finalized by the Akka team. Given the need to provide some form of outside fault tolerance, we settled for the Eventsourced persistence library [Krasser 2013]. Though the *Akka persistence* library is inspired by the Eventsourced library, the Eventsourced library contains certain programming difficulties which have been resolved by Akka.

For example, when creating a persistent actor in Eventsourced, the persistent actor must be labelled with a unique number. A problem arises in keeping track of which actor is assigned which number, which becomes more difficult as the number of actors in different branches grow. The programmer has to be mindful that a number given to a certain actor is not also assigned to another actor. This issue arose in our implementation, it became difficult to maintain the persistent actor numbers when allowing the user to define the number of routees. In the case of the *Akka persistence* library, it does not require that a persistent actor be associated with a unique number. Given the above dilemma, a decision was made to use the Eventsourced persistence library on actors that required their state to be maintained after an actor restart. This minimized the amount of actors using Eventsourced, simplifying the number assignment of each persistent actor.

5.2.6 System Configuration

The network application has a number of attributes that can be configured in order to improve performance. The Akka configuration is used to change the default behaviour of the Akka system or adapt the Akka system for a specific runtime [Typesafe 2013]. For example, attributes that can be configured include the default mailbox type, actor remoting, tunning of routers and other attributes. In the network application, the Akka configuration is currently only used for logging properties. Initially, the Akka configuration was also used to declare the lower bound and upper bound of routee instances a router can create. But due to using a custom supervision strategy for the routers, a separate configuration file had been used. The routee's lower bound and upper bound instances are declared in a separate user defined Extensible Markup Language (XML) file(s). The files also configure the amount of time an actor will wait for a response from a *future* message.

5.2.7 Communication Format Between Nodes

JSON is used to exchanged data between nodes because it offers a well-ordered and simple way of organizing information. JSON is a light weight data-interchange format, which is easy for humans to read and write [Crockford 2006]. Though Scala includes a JSON library, through investigation on various forums it was noted that the Scala JSON parser is slow compared to other JSON parsers [Chen-Becker

2011]. The JSON library in Scala was created in the early days of Scala and hasn't been updated since, because several JSON libraries have been created that outperform the Scala JSON library. Lift-JSON is a library, from the Lift web framework, which provides utilities for parsing and formatting JSON data [del Pilar Salas-Zrate *et al.* 2012]. Besides the performance of parsing JSON data, we chose Lift-JSON because of its ease of use in constructing JSON messages in our code.

5.3 System-Coordinator

A front-end interface is required to interact between the network application functions, the business logic and hosts. The System-Coordinator acts as the required interface. Messages directed to the network system, using a specific Uniform Resource Identifier (URI) are captured by the System-Coordinator. The System-Coordinator proceeds to forward the message to the appropriate service.

Socko is an open-source Scala based web server that uses Netty's NIO networking framework and Akka's actor paradigm [Community 2013; Imtarnasan *et al.* 2012]. By embedding Socko into our Scala based network application, Socko exposes the actors in the application as HTTP and WebSocket end-points. Incoming messages received by Socko are wrapped within a SockoEvent. A SockoEvent is used to read incoming data and write outgoing data. SockoEvent's are passed onto the Socko Routes, which through a set of rules dispatch incoming events to specific actors. Socko uses pattern matching extractors to decide which actor the Socko Route should pass the SockoEvent to. Socko currently supports the following pattern matching extractors:

- Event matching by matching the type of SockoEvent e.g., WebSocket Handshake
- Host matching by matching the host name received in a HTTP request that triggered the Socko-Event e.g., www.mydomain.com
- Method matching by matching the method received in a HTTP request that triggered the Socko-Event e.g., GET
- Path matching by matching the path received in a HTTP request that triggered the SockoEvent e.g., /client/login
- Match by using the query string received in a HTTP request that triggered the SockoEvent e.g., action=save

Socko also allows the usage of two or more Route extractors to match a specific pattern. Given that Socko uses Netty's NIO networking framework, Socko implements Netty's capability to use HTTP and SPDY transparently [Lee 2012]. To use SPDY in Socko the Next Protocol Negotiation (NPN), that negotiates the protocol, is used in a manner that avoids additional round trips [Langley 2012].

5.4 Storage

The storage component consists of actors that interact with the Voldemort storage. Figure 5.3 displays the design overview of the storage component. Figure 5.4 illustrates the sequence of actions when interacting with the storage component.

StorageCoordinator

The StorageCoordinator is the top level actor in the storage block of the network system. The StorageCoordinator actor receives WebSocket transmissions from the System-Coordinator containing JSON content. The actor then decrypts data contained in the message allowing the actor to route the message to the corresponding child actor for the request given in the JSON message.

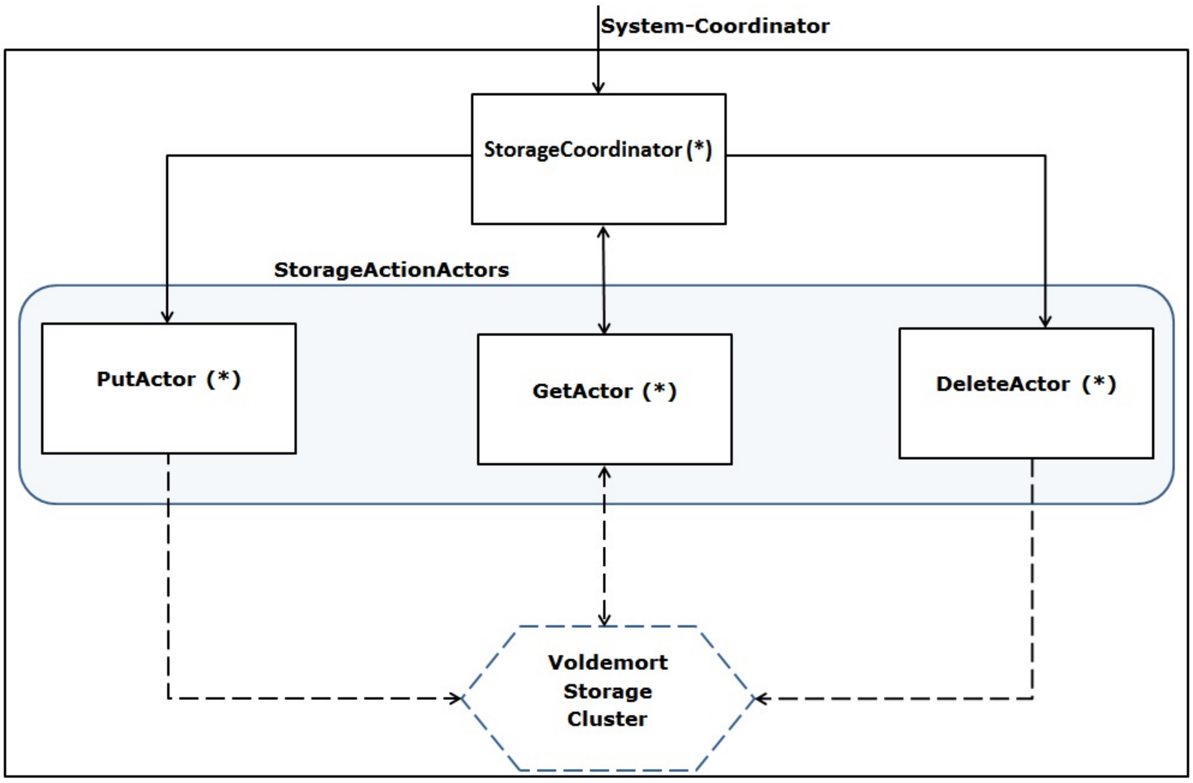


Figure 5.3: Design overview of the storage component.

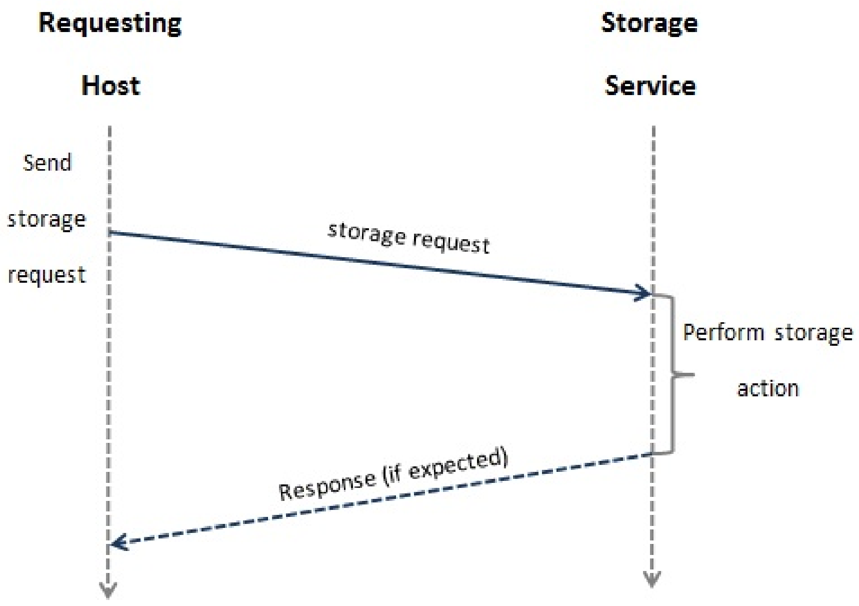


Figure 5.4: Storage component state diagram.

StorageActionActors

The StorageActionActors block consists of actors that interact with the Voldemort storage and perform storage operations. The PutActor uses the key and value given in the JSON message to create data on the Voldemort storage. If the key already exists on the Voldemort storage the value associated with the key is overwritten with the new value given in the JSON message. The DeleteActor deletes the data associated with the key given in the JSON message. The GetActor asynchronously retrieves data from the Voldemort storage associated with the key in a given period of time. Once the value is retrieved from the Voldemort storage, the value is passed back to the StorageCoordinator actor, which then returns the value to the host that requested the data. If the GetActor failed to retrieve the data in the given time, noted in the configuration file, the StorageCoordinator actor returns a timeout message to the client. The get operation between the StorageCoordinator and GetActor is non-blocking and asynchronous, which allows the StorageCoordinator to perform other operations while waiting on a response from the GetActor.

5.5 Messaging

The messaging component consists of actors that manage the publish-subscribe operations. A host publishes a message without explicitly specifying the recipients or having knowledge of the recipients. The recipients are subscribed to a logical topic that delivers a specific class of messages. Figure 5.5 displays the design overview of the messaging component.

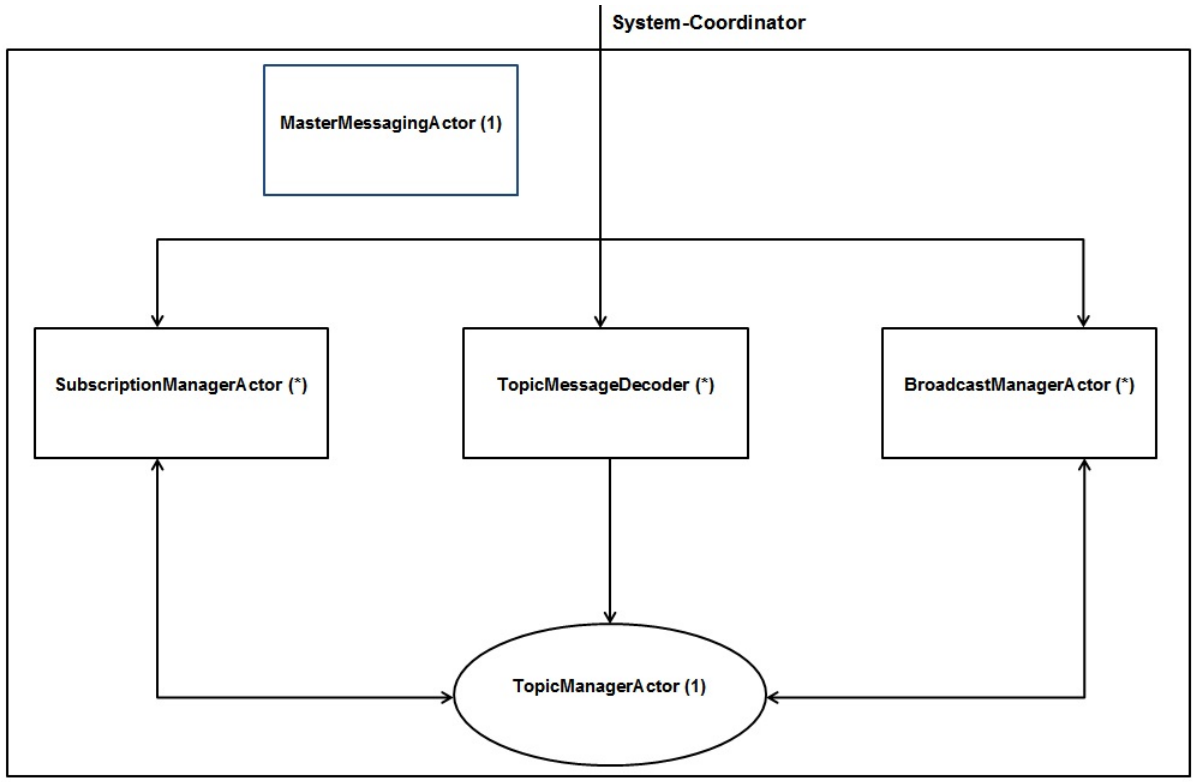


Figure 5.5: Design overview of the messaging component.

MasterMessagingActor

The MasterMessagingActor is the top level actor in the messaging block. The MasterMessagingActor simply creates the other actors in the messaging block according to the configuration file. The MasterMessagingActor also supervises all the child actors it created.

TopicMessageDecoder

The TopicMessageDecoder actor receives JSON messages from the System-Coordinator which are meant to create or delete a topic. The TopicMessageDecoder decodes the JSON messages and sends the decoded message to the TopicManagerActor.

TopicManagerActor

Socko allows us to create a special actor, known as a WebSocketBroadcaster, which sends data to all the clients who have registered to receive data from the actor. Messages received from the TopicMessageDecoder actor are assigned to create or delete a topic. If the message directs the TopicManagerActor to create a new topic, the TopicManagerActor will first check if the topic exists. If the topic already exists, the TopicManagerActor sends a message back to the host that requested the topic, stating that the topic already exists. If the topic does not exist, a WebSocketBroadcaster actor is created and inserted into a map, linking it to the topic name. An Eventsourced message is then sent to the TopicManagerActor, directing the actor to take a snapshot of its current state. A message is also sent to the client stating that the topic was successfully created.

SubscriptionManagerActor

When a host wishes to subscribe to a specific topic, it must establish a connection with the WebSocketBroadcaster specific to that topic by using a URI. During the WebSocket handshake, the SystemCoordinator passes the handshake data to the SubscriptionManagerActor. The SubscriptionManagerActor checks if the topic exists. This is done by sending an asynchronous request to the TopicManagerActor. If the topic does exist, the SubscriptionManagerActor receives the WebSocketBroadcaster actor related to the topic. The SubscriptionManagerActor then proceeds to authorize the WebSocket connection and register the host to receive messages broadcasted for the topic. If the topic which the host wished to subscribe to does not exists, the SubscriptionManagerActor does not authorize the WebSocket connection. When a host wishes to unsubscribe from the topic, the subscribed host can simply disconnect the WebSocket.

BroadcastManagerActor

A host that wishes to publish a message to other hosts subscribed to a topic simply needs to send text data through its WebSocket that is registered to the relevant WebSocketBroadcaster. Once the SystemCoordinator receives the data it passes it onto the BroadcastManagerActor. The BroadcastManagerActor uses the URI from the WebSocket connection to retrieve the topic's WebSocketBroadcaster by sending an asynchronous request to the TopicManagerActor. When the BroadcastManagerActor has obtained the relevant WebSocketBroadcaster, it sends the text data from the client to WebSocketBroadcaster. The WebSocketBroadcaster then proceeds to publish the message to all the subscribed hosts. The host that published the message will also receive the same the message. Figure 5.6 shows the state diagram for a host to subscribe to a topic and publish a message.

5.6 Service Routing

The service routing component consists of actors that manage connections to external services outside of the network application. It also routes messages between hosts and the external services. Figure 5.7 presents the design overview of the service routing component. The service routing block is divided into the three following blocks: ServiceRegistrar, ServiceRequest and ServiceResponse. The blocks were created on the premise that a service must first register the service it is offering to the network. Once the service is registered, other hosts can make requests. Moreover, the service may be required to respond to the requesting host.

MasterServiceActor

The MasterServiceActor is the top level actor in the service routing block. The MasterServiceActor creates and supervises the following actors: ClientRequestServiceActor, ServiceDispatcherActor, ClientWebsocketLogActor and the AdminServiceActor.

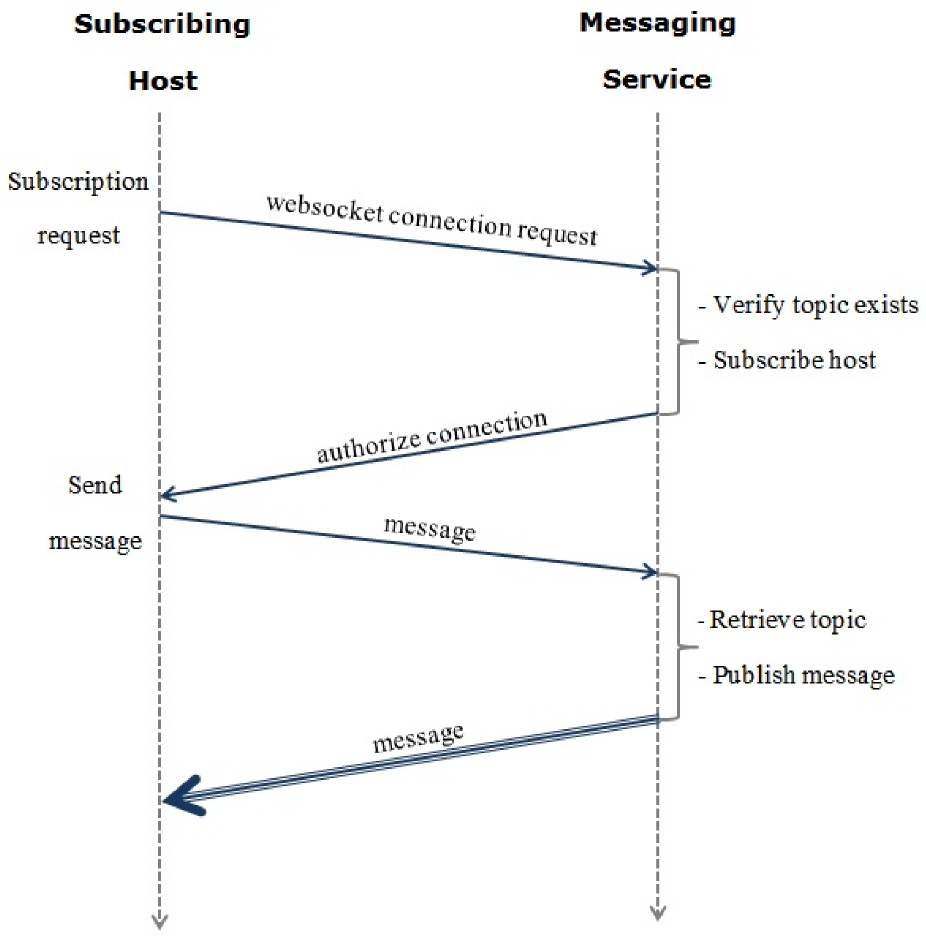


Figure 5.6: Subscribe and publish messaging state diagram.

ServiceRegistrar

The ServiceRegistrar block consists of actors that record and maintain the list of services available. When a service wishes to register or deregister its service, the AdminServiceActor needs to be notified. This is done by directing a JSON message to the ServiceDispatcherActor from the System-Coordinator, along with the WebSocket channel the service established the request with. When the ServiceDispatcherActor receives a message it decodes the JSON message. The JSON message received contains instructions for the ServiceDispatcherActor and data in the form of another JSON message. The instructions indicate if the ServiceDispatcherActor should forward the data to the ResponseDecoderActor or the AdminDecoderActor. If the data is to be forwarded to the AdminDecoderActor, the AdminDecoderActor will further decode the JSON data and then pass the data to the AdminServiceActor. The AdminDecoderActor would have also received the WebSocket channel of the service. The WebSocket channel is passed along with data to the AdminServiceActor.

The AdminServiceActor maintains a map of all the services which have been registered for the network application. By using a unique key, related to the service, the system is able to retain a connection to each service. Messages which the AdminServiceActor receives from the AdminDecoderActor indicate whether a service is to be registered or deregistered. If a service is to be registered, the WebSocket channel is inserted into the map using the unique value contained in the message as the key. A message is then sent to service indicating it was successfully registered. If the service wishes to deregister, the WebSocket channel is removed from the map (along with the key) and a message is then sent to service indicating it was successfully deregistered. Once the service receives the message that service has been deregistered, the service can disconnect the WebSocket. In both cases of registration and deregistration,

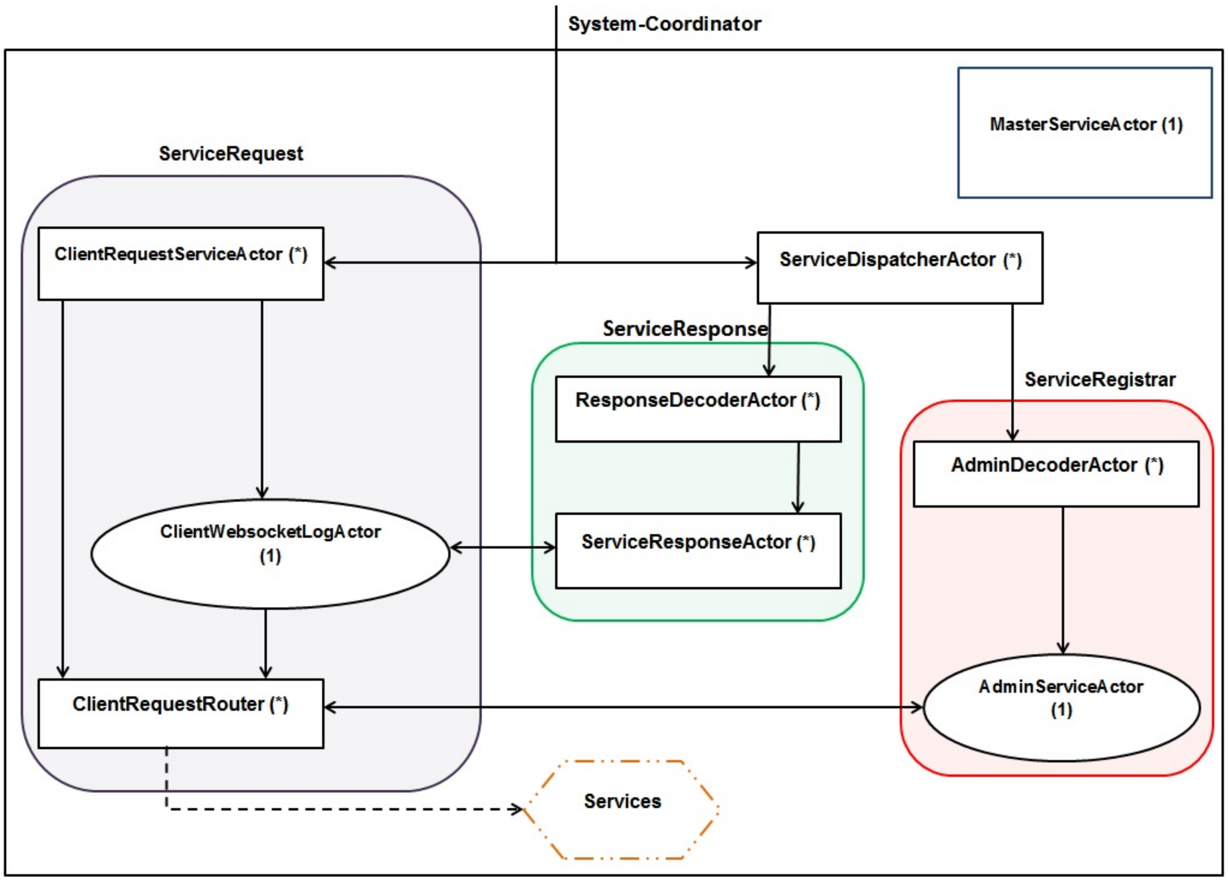


Figure 5.7: Design overview of the service routing component.

the AdminServiceActor takes a snapshot of itself after performing each operation.

ServiceRequest

The ServiceRequest block contains actors that handle requests (or updates) from clients that require a specific service. The ServiceRequest block finds the requested service and routes the request to the service. The ClientRequestServiceActor receives JSON messages from the System-Coordinator, which it proceeds to decode. The data obtained in the JSON message will indicate if the client expects a response from the service. If a response is expected, the data is passed onto the ClientWebsocketLogActor; if no response is expected, the data is sent to the ClientRequestRouter. The ClientWebsocketLogActor records the client's WebSocket channel in a map using a unique key obtained from the WebSocket constructor. Once the WebSocket channel has been recorded the data obtained from the JSON message is passed onto ClientRequestRouter along with the key associated with the clients WebSocket channel. The ClientRequestRouter asynchronously retrieves the WebSocket channel associated with the service from the AdminServiceActor. Once the ClientRequestRouter receives the WebSocket channel associated with the service, the ClientRequestRouter sends the data obtained from the JSON message to the service.

ServiceResponse

The ServiceResponse block handles response messages for hosts expecting a response from a service. The same WebSocket channel that the service is registered under is used to obtain a response from the service. We stated earlier on that the ServiceDispatcherActor receives messages from the System-Coordinator that are sent from services. Messages that contain a response for a client are directed to the ResponseDecoderActor. The ResponseDecoderActor further decodes the JSON data contained in the message and passes the data to the ServiceResponseActor. The ServiceResponseActor sends an asynchronous request to the ClientWebsocketLogActor for the clients WebSocket channel. Once the ServiceResponseActor obtains the WebSocket channel, the response message is sent to the client. The ServiceResponseActor also sends an instruction message to the ClientWebsocketLogActor. The message directs the ClientWebsocketLogActor to remove the clients WebSocket channel from the map. Figure 5.8 illustrates the sequence of actions when a host makes a request from a service.

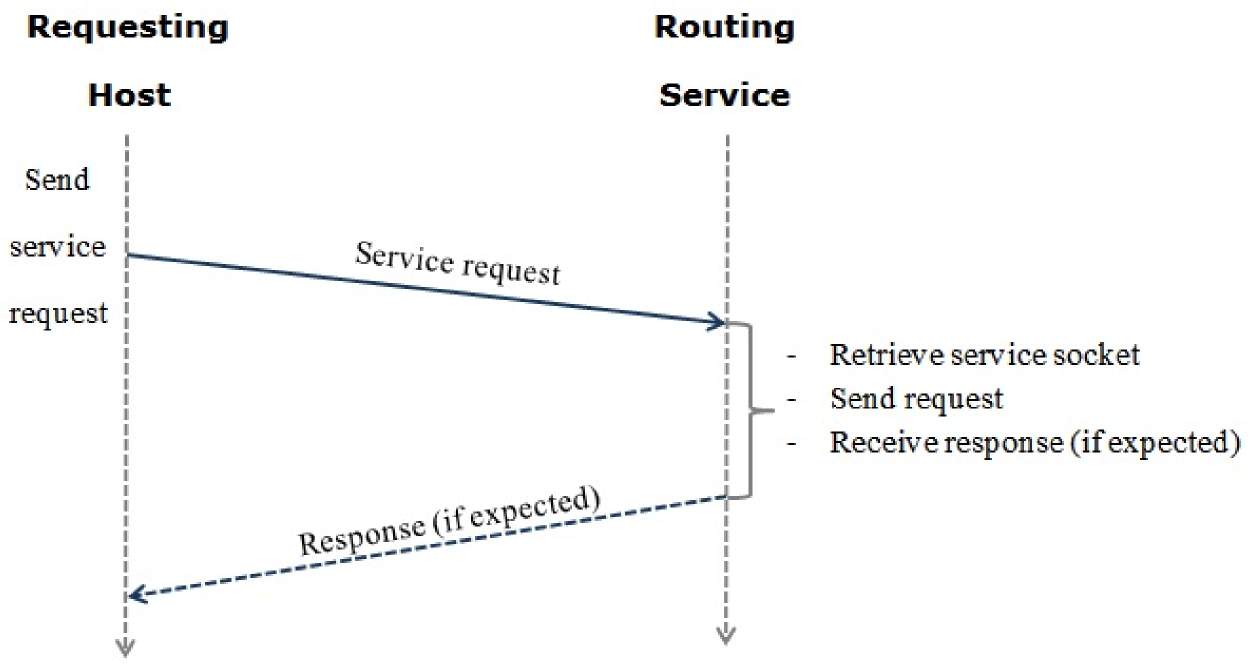


Figure 5.8: Service request state diagram.

5.7 Supernode System

In Section 3.4 of Chapter 3, we decided that a supernode infrastructure will be used. The Supernode Control protocol is able to deliver messages in an adequate time frame and is able to maintain local consistency in a region of the game world. In the supernode solution, each node contains the storage, messaging and service routing service. Along with the intention of managing the game state of a region in the game world, the supernode also serves to relay messages between the nodes. Figure 5.9 gives an overview of the supernode design. The sub-nodes establish a connection with the supernode using Jetty. Jetty allows a host to establish a SPDY connection or a standard HTTP/HTTPS connection.

5.8 Instructions For Running

The peer-to-peer application network can be found on the following site: Sharingan Project [1]. The project was managed with Simple Build Tool (SBT), an automation build tool for Scala and Java, similar to Apache Maven and Apache Ant [Harrah 2013]. The peer-to-peer system can be run within SBT, but for portability it is best to create a *fat JAR*. A *fat JAR* is a single executable jar file, which not only contains the executable application but also contains all the dependencies and referenced libraries. The SBT-assembly plugin was used to build the fat JAR executable [SBT 2013].

[1]https://github.com/Bongani/sharingan

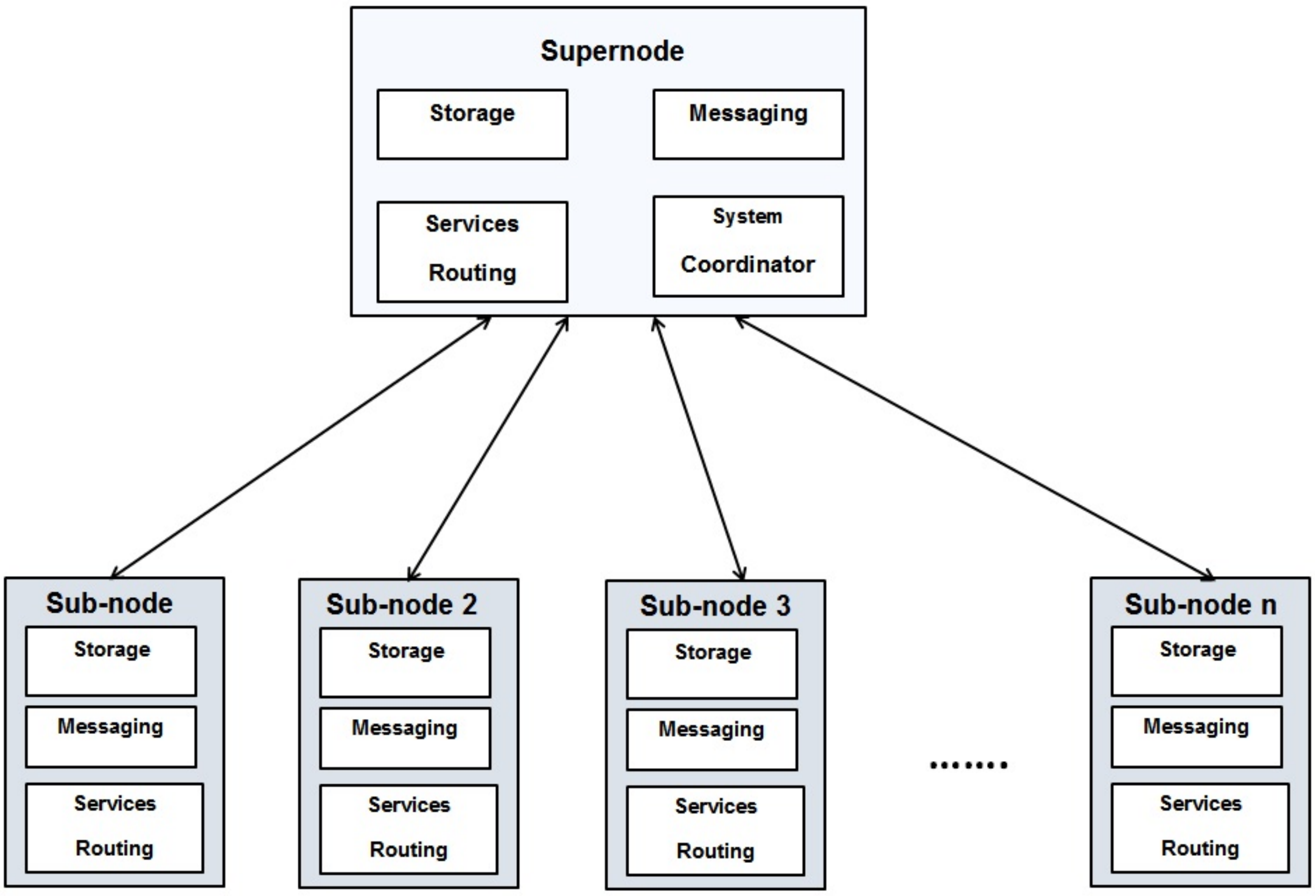


Figure 5.9: Overview of the supernode system.

5.9 Conclusion

In this chapter, we presented in detail the design and architecture of our peer-to-peer gaming solution. The chapter focused on the design of the three central aspects of our system: persistent storage, message dissemination and service routing. A front-end Akka based service, Socko, was also introduced. Socko is used to route incoming messages to the correct function. The overall Akka attributes used in the system and how each attribute effects the performance of the system was discussed. The next chapter examines the performance of our peer-to-peer solution, evaluating the communication protocol, SPDY, and the time delay in processing network messages.

Chapter 6

Testing and Analyses

6.1 Introduction

In the previous chapter, we presented the design of our system. In this chapter we assess whether our peer-to-peer gaming solution can reliably deliver messages in a timely manner. The evaluation process contains seven sections. The first test, in Section 6.2, evaluates the communication protocol between nodes. We perform analysis on the response time when SPDY is used. We use the results to determine if it is possible to reduce the latency on the reliable Transmission Control Protocol. In Section 6.3, the front-end service, Socko, is benchmarked against web servers that can be used to host online games. The results allow us to determine if Socko provides adequate performance compared to current web solutions that handle multiple users at a time. In Section 6.4, we evaluate the actor-based network solution to measure the response time for processing messages. The network solution is evaluated under three different circumstances in sub-sections 6.4.2, 6.4.3 and 6.4.4. Sub-section 6.4.2 studies the response time of the system under high user loads with varying number of actors handling the load. By increasing the number of actors to complete a certain task, we examine if the system's event-driven actor paradigm can reduce the time taken to process messages retrieved from the network. Sub-section 6.4.3 studies the latency of the system under varying user loads to determine when the performance of a service decreases. Sub-section 6.4.4 analyses the system's response time during actor failure to examine the effect of component failure. Taking from the results obtained in Section 6.4, Section 6.5 compares the individual performance of each actor in the system. The comparisons provide insight into identifying bottlenecks within the system. Section 6.6 evaluates the response time of the network gaming solution in a distributed environment. The results obtained determine if the peer-to-peer solution can scale as more nodes are introduced into the network. Section 6.7 concludes the chapter by discussing how the results obtained in this chapter answer the research questions formulated in Chapter 3.

6.2 Communication Protocol Evaluation

Many MMOG require that the time taken to transport a message over the network be minimized in order to present a highly interactive game world with no lag. In Chapter 3, we proposed the notion of using SPDY to reduce the delay experienced on the reliable Transmission Control Protocol. In this experiment, we compare the time taken to download multiple items from the HTTP and SPDY communication protocols. The test indicates whether our assumption in using SPDY as a communication protocol improves message delivery time upon the reliable TCP transport layer. To compare the latency between SPDY and HTTP we used the web performance tool, WebPagetest. WebPagetest is an open-source tool developed and supported by Google that measures and analyses the performance of web page load times [Meenan 2012]. The WebPagetest tool can analyse web pages from different locations, using several web browsers, under different Internet connection speeds and other user specified conditions.

6.2.1 Evaluation Methodology

SPDY uses the TLS encryption to provide a secure connection between machines. Similar to HTTPS, SPDY requires additional round trips to setup a secure connection compared to a standard HTTP connection setup. When a HTTPS connection has been established, there is no difference in packet delivery time between HTTPS and HTTP [Sissel 2010]. In order to review the effect the connection setup has on latency, SPDY was compared to HTTP and HTTPS. The evaluation was conducted in two environments, the first being in a private local environment which we created our own WebPagetest client. The second set of tests was conducted in a public environment using a WebPagetest client in London. The private instance removed network activity that could influence the results. The public environment test displayed real world results. The experiment consisted of measuring the load time from a web page that we created. The web page consisted of text, a Cascading Style Sheets (CSS) file and 230 Portable Network Graphics (PNG) images. The image sizes were all 48×48 pixels. The amount of requests simulated the number of requests a game server would be required to serve at a single point. Given that the Socko web server provides HTTP, HTTPS and SPDY communication protocols, Socko was used to host and serve the web page. The private environment tests were conducted eight times from which we obtained the average download time. The public test was also conducted eight times, but on four different occasions during the day to obtain the average download time and to diminish the effect of network traffic during different times of the day. Two tests were conducted in the morning, two tests were conducted in the afternoon, two tests were conducted in the early evening and another two tests were conducted at midnight.

6.2.2 Environment

Below, we list the specification of the environment in which this experiment was conducted.

WebPagetest configuration
The WebPagetest client used in the experiments was set to the following configurations:

- Browser: Google Chrome
- First view only
- SSL certificate cache cleared on each run
- SSL certificate errors/warnings ignored
- Dial-up connection: 49 Kbps downlink, 30 Kbps uplink
- Digital Subscriber Line (DSL) connection: 1.5 Mbps downlink, 384 Kbps uplink
- Cable connection: 5 Mbps downlink, 1 Mbps uplink

WebPagetest private client
The machine used to host the private Web Pagetest client instance had the following specifications:

- Intel Core 2 Duo T5800 @ 2.00GHz
- 2 GB RAM
- 300 GB available hard drive storage capacity
- Windows 7 Ultimate (x86)

Web server
Below, we indicate the specifications of the machine used to host the Socko web server, containing the web page requested by the WebPagetest client:

- Intel Core i5-2410M @ 2.30GHz

- 4 GB RAM
- 600 GB available hard drive storage capacity
- Ubuntu 12.04 (precise) 64-bit
- Kernel Linux 3.2.0-29-generic
- File descriptor: 512 000 for soft value; 1 024 000 for hard value
- Socko: version 0.3.1; Akka 2.2.0; Java 1.7.0_10 (3 GB heap memory)
- In the public test, the Internet connection used had a 8 Mbps downlink and a 2 Mbps uplink.

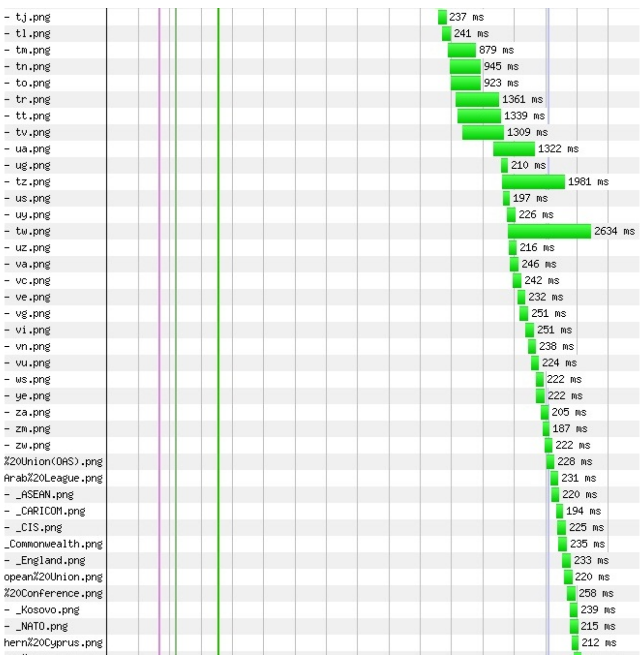
6.2.3 Results and Analysis

Table 6.1 lists the average web page download times achieved by the HTTP, HTTPS and SPDY communication protocols in different environments. In the case where the test was conducted in a private instance, SPDY exhibited a faster download time than HTTP and HTTPS. As better link connections were emulated, the HTTP and HTTPS communication protocols showed a remarkable improvement in download time, diminishing the improvement SPDY initially showed. In the public network test, the HTTP communication protocol showed a better load time than SPDY and HTTPS on the dial-up connection. Using the DSL and cable connections, SPDY exhibited a faster download time than the HTTP and HTTPS communication protocols. In the dial-up connection test, HTTP was able to experience an initial faster download time due to the secure connection procedure SPDY and HTTPS endure. The connection setup was slow enough to affect SPDY's overall performance. In the cable and DSL tests, the speedup in SPDY mitigated the time it took to establish a secure connection. From the results obtained from Table 6.1, SPDY's improvement on a network with no network traffic is minimal, given a better line connection. In an environment with alternating network traffic, SPDY provides a significant reduction in network delay on the TCP transport protocol. SPDY is less likely to provide any significant improvement to games hosted on local area networks (LAN), but can reduce the delay of messages for games hosted across the Internet.

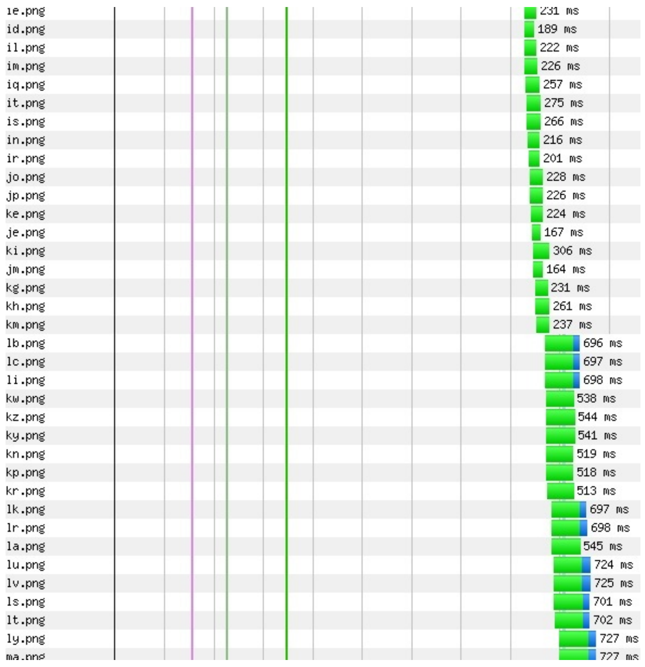
	Private			**Public**		
Connection	Dial-up	DSL	Cable	Dial-up	DSL	Cable
HTTP	99.2	4.09	2.02	112.71	18.5	17.43
HTTPS	101.02	5.41	2.43	121.67	21.24	20.37
SPDY	92.95	3.89	2.01	117.38	16.05	14.96

Table 6.1: Average download times (seconds) of different communication protocols in a private and public environment.

Figure 6.1 displays screenshots of the download waterfall for HTTPS and SPDY. The waterfalls were taken from the public test environment. In the SPDY waterfall, we note that multiple items were downloaded simultaneously, whereas in the HTTPS waterfall, items were downloaded in a small faction. SPDY's multiplexing stream property allows for transferral of multiple items over a single TCP connection at any single point. Though it seems that the faction of items are downloaded more rapidly in HTTPS (due to a less congested link), acknowledgements and requests of the next set of items still need to be made once an item has been received by the client. The practice of downloading a single item at a time was the attribute that contributed to HTTP and HTTPS poor performance against SPDY, especially when a delay occurred whilst downloading an item. In Figure 6.1a, the HTTPS waterfall, we note several items which take longer to download. Specifically, the *tw.png* image which took 2634 milliseconds to download. The download delay experienced by these items had an impact on the request and retrieval time of other items.



(a) HTTPS waterfall



(b) SPDY waterfall

Figure 6.1: A comparison of the HTTPS and SPDY transfer schemes taken from the public network test.

6.3 Front-end Service Evaluation

In this experiment, we compare our gaming network to existing online gaming solutions. These comparisons provide insight into our gaming solutions performance. Most of the peer-to-peer gaming networks we researched were simulations of peer-to-peer gaming communication models; in essence, no real working network existed. The network models which did have an existing peer-to-peer gaming network had legacy issues. These networks were dependent on specific library versions which were no longer obtainable. VAST was the only peer-to-peer gaming network which was available and working [Hu *et al.* 2006]. Unfortunately the VAST code was tied to its game demonstration package, which made it difficult to use for our evaluation. Through further investigation we discovered that several web servers were suitable to host multiplayer online games. Comparing our systems' front-end service, Socko, against web servers that have online multiplayer gaming capabilities allowed us to gauge our systems response time to requests against varying user loads. During game play, static data (such as sprites) and dynamic information (such as player position) are exchanged between clients and servers. The above criterion was used to choose the following three web servers for the experiment:

- **Node.js**

  Node.js is an event-driven JavaScript environment for developing network applications. Node.js is based on Google's V8 runtime implementation, focusing on low memory and high performance non-blocking operations [Tilkov and Vinoski 2010]. The Node.js event loop is handled by a single thread. The event loop is not exposed to the developer, therefore properties such event notification are hidden [Erb 2012]. Though Node.js is commonly used for server side applications (such as, web servers), it can also be used to create client side applications (for example, a chat client). Despite Node.js being a fairly new platform, it has a large library containing numerous plugins. Node.js has been used for various applications such as web-based communication tools and gaming applications [Crane 2013].

- **Nginx**

  Nginx is an open-source high-performance web server, and reverse proxy for HTTP, HTTPS, SMTP, POP3 and IMAP protocols [Brown and Wilson 2011]. Nginx uses a non-blocking event-based model to handle requests. Nginx excels at serving static content, but does not show the same high performance for serving dynamic content [Schroeder 2013; ServerStack 2012]. Nginx allows requests to be handled by a backend server, in other words, Nginx becomes a reverse proxy. This enables Nginx to deliver static content swiftly, and pass on dynamic requests over to a backend server that can respond to dynamic content swiftly. The reverse proxy service also allows for load balancing, distributing requests to multiple backend servers. Nginx can also act as a reverse caching proxy, reducing the amount of requests to be processed by a backend server.

- **Apache HTTP Server**

  The Apache HTTP Server, commonly referred to as Apache, is currently the most popular web server [Netcraft 2013; W3Techs 2013]. Some of the features provided by the Apache server include support for CGI (Common Gateway Interface), SSI (Server Side Includes), URL redirection, user authentication, proxy caching abilities, additional module support and so forth [Foundation 2010]. Apache is a thread-based web server, spawning a new thread for each request. Apache has been shown to be slower than Nginx when serving static content, but faster with serving dynamic content [Schroeder 2013]. In order to improve performance, Apache has implemented a Multi-Processing Module (MPM) that mixes the use of several processes and threads per process.

6.3.1 Evaluation Methodology

Apache JMeter, commonly referred to as JMeter, is an open-source tool used to simulate load on a server, network or object; testing its strength or analysing the overall performance under different load types [Foundation 2013]. JMeter can be used on web application servers such as Mail servers, FTP servers, and SOAP services, amongst other servers. Apache JMeter also allows the use of community developed plugins which increases the number of applications JMeter can benchmark.

The Apache JMeter tool was used to simulate user requests on the web servers. Each simulated user in JMeter requested a web page from the server four times. The test was repeated four times to give an average response time and to remove the presence of unforeseen network activity that could negatively influence the results. The web page used consisted of text, a Cascading Style Sheets (CSS) file, twenty images of varying sizes and a single dynamic script. The Apache, Nginx and Socko web servers can be configured to optimize performance of each web server. Despite this, default configurations of each web server were used. Configuring a web server is dependent on many parameters such as the hardware capabilities of the machine. We wished to avoid unintentionally configuring one web server to perform better than the others, thus predetermining the results.

6.3.2 Environment

Below, we list the specifications of the environment which this experiment was conducted in. The web servers were installed on the same machine, but we ensured that only one web server was operational at a single point to avoid resource consumption from the other web servers.

Webserver node:

- Intel Core i7 @ 3.40 GHz - 3770
- 8 GB RAM
- 500 GB hard drive storage capacity
- Ubuntu 12.04 (precise) 64-bit
- Kernel Linux 3.8.0-29-generic

- File descriptor: 512 000 for soft value, 1 024 000 for hard value
- Apache: version 2.4.7
- Nginx: version 1.5.6
- Node.js: version 0.10.24
- Socko: version 0.3.1, Akka 2.2.0, Java 1.7.0_10 (7GB heap memory)

Apache JMeter node:

- Intel Core i5 @ 2.00 GHz
- 2 GB RAM
- 100 GB hard drive storage capacity
- Apache JMeter 2.10 (1GB heap memory)
- Ubuntu 12.04 (precise) 64-bit
- Kernel Linux 3.8.0-29-generic

6.3.3 Results and Analysis

Table 6.2 shows the average response time, in milliseconds, for different user loads against different web servers. The results are graphed in Figure 6.2. From the results, it is immediately evident that Nginx had the best performance compared to all other web servers. Nginx's ability to handle high load request in a short period is likely one of the reasons it can be used as a reverse proxy engine. The Socko web server performed on par with the Apache web server. The Node.js web server had the poorest performance with high load users, but had one of the fastest response times with low user numbers. Node.js performance is said to lie within serving dynamic content [Virkki 2013]. Though the web page used contained some dynamic content, it was fairly populated with static content. This could have led to Node.js exhibiting poorer performance. Given that Socko's performance matches Apache's performance, currently the most popular web server, Socko is therefore able to handle high user loads for an online game.

Number of users	**Apache**	**Nginx**	**Node.js**	**Socko**
1000	33	10	12	18
2000	34	12	18	28
3000	36	17	47	32
4000	44	20	92	38
5000	81	37	126	89
6000	115	49	364	91
7000	143	77	502	125
8000	259	102	623	209
9000	407	130	815	359
10000	709	152	1045	630

Table 6.2: Average response times for Apache, Nginx, Node.js and Socko webservers in milliseconds against various user loads.

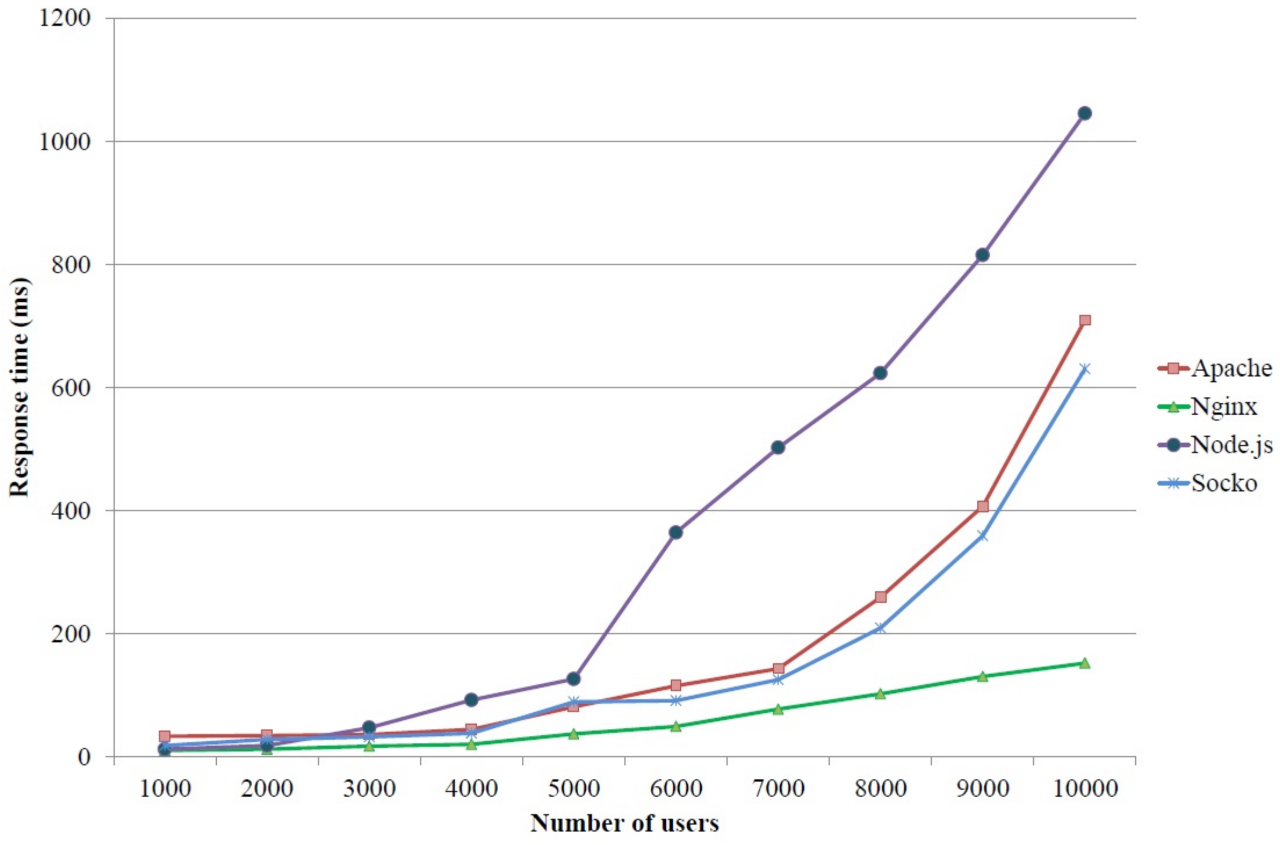


Figure 6.2: Average response times for request made to Apache, Nginx, Node.js and Socko web servers versus increasing users.

6.4 Actor System Evaluation

In a MMOG, the time taken to process a large amount of messages retrieved from the network must be kept to a minimum in order to avoid game play lag. In Chapter 3, we proposed using an event-driven paradigm as it provides higher concurrency compared to thread-based systems. We adopted the Akka actor toolkit to develop our system. Akka is designed to support fault-tolerant, low latency and high throughput applications. Also Akka allows us to distribute the work load by creating multiple instances of a specific actor, and thereby reducing the amount of time taken to process a batch of messages. In the following experiments, we measured the time taken for messages to be processed by our network application. The experiments measured (i) the actor scalability of the network application, (ii) the response time during varying user loads, and (iii) the effect that a component's failure has on the response time. Each of the experiments consisted of evaluating the three functions that our system provides: the storage component, the messaging component, and the service routing component. On each test, requests from a numbers of users were sent to each specific component. We measured the time taken for the message to traverse the module and respond to the requesting user. The description on how each module was configured and tested is given below.

Storage component

The analysis of the storage component was based on the procedure of retrieving data from the Voldemort storage. In context of the storage component tasks, the get procedure returns a message to the requesting host. Furthermore, we separated the test on the storage module into two phases. The first phase tested the storage module as it is, retrieving values from the Voldemort storage cluster. The second phase consisted of disabling the connection to the Voldemort storage. Instead, when the GetActor receives a request to retrieve data, the GetActor responds with a random message. The test allowed us to establish the time difference in which a message should traverse the storage component and the time taken to retrieve a message from the Voldemort storage. The following experiments were solely concerned with the responsiveness of the actor system, therefore, only a single Voldemort store node was used during this phase of the analysis. 200,000 randomly generated data values were stored on the Voldemort storage system. Users were simulated through Apache JMeter to retrieve a value from the Voldemort storage. Each user requested a value from the storage four times to obtain a justifiable average response time.

Messaging component

The analysis on the messaging component was centered on measuring the time taken for a user to subscribe to a topic and broadcast a message. To analyse the time taken to publish and receive a message, a topic was created that users subscribe and publish messages through. Users were simulated through Apache JMeter. Each simulated user published a message four times after subscribing.

Service Routing component

In this experiment, we measure the time taken for a message to traverse the service routing component. We do not measure the response time from a specific service. Given that a service is hosted outside our gaming network solution, the response would be dependent on the service and no longer just the peer-to-peer gaming solution. Also, the performance of each service application can vary; therefore, we cannot obtain a general performance standard from a single service application. The experiment provides a prediction on the response time, solely based on the service routing component. The following amendments were made to the service routing component: When the ClientRequestRouter receives a request message, the message is sent to the ServiceDispatcherActor, instead of sending the message to a registered service. The ServiceDispatcherActor proceeds to process the response for the host in the manner described in Section 5.6. Similar to the tests described above, users were simulated using Apache JMeter, with each user making four requests.

6.4.1 Environment

Below, we list the specifications of the machines which were used during this experiment.

Gaming solution node:

- Intel Core i7 @ 3.40 GHz - 3770
- 8 GB RAM
- 500 GB hard drive storage capacity
- Ubuntu 12.04 (precise) 64-bit
- Kernel Linux 3.8.0-29-generic
- File descriptor: 512 000 for soft value; 1 024 000 for hard value
- Akka 2.2.0
- Socko 0.3.1
- Java 1.7.0_10 (7 GB heap memory)

Voldemort node:

The Voldemort storage was placed on a machine with the same specifications as the node indicated above. The Voldemort storage node was allocated 7 GB of JVM memory heap. The following configurations were used for Voldemort:

- Voldemort version 1.3.0
- Persistence storage engine: BDB
- Routing policy: client
- Replication factor: 1
- Required read and writes: 1
- Number of partitions: 1

Apache JMeter node:

Five nodes were used to generate the simulated users from JMeter. A sixth node serves as the master node for the JMeter instances across the five nodes, coordinating the tests amongst the nodes and collecting the results from each node. Each node consisted of the following specifications:

- Intel Core i5 @ 2.00 GHz
- 2 GB RAM
- 100 GB hard drive storage capacity
- Ubuntu 12.04 (precise) 64-bit
- Java 1.6 (1GB heap memory)
- Apache JMeter 2.10

6.4.2 Actor Scalability

This experiment examined the scalability of the actor-based peer-to-peer gaming system. Some of the actors in the system are created from router actors, allowing for multiple instances of a specific actor. The premise behind this was that the router actor can distribute the incoming messages to several actors (routees), and in turn this would distribute the work load and decrease the amount of time messages spend in a mailbox. The purpose of this test was to determine if increasing the number of actors in the system improves the message response time. In the storage component, the StorageCoordinator routees were increased, this in turn also increased the number of StorageActionActors due to the hierarchical structure of the storage component. In the messaging and service routing components, all the routees were increased. 25,000 users were simulated through JMeter. The test was conducted four times to obtain the average response time.

Results and Analysis

Table 6.3 shows the response times for a nodes services against the amount of routees each service uses to fulfil requests. Figure 6.3 graphs the results. Except for the messaging service, it is evident that increasing the number of routees decreases the response time. Take for example the actors responsible for the retrieval of data from the Voldemort store. Simply by increasing the number of routees in the storage component, the response time decreased dramatically (79% improvement factor from 1 actor to 10 actors). Instead of having a single actor performing a request on the Voldemort storage, multiple actors were able to take advantage of Voldemort's NIO non-blocking sockets and respond to requests simultaneously. As more actors were used for retrieving data from the Voldemort storage, the response time grew closer to that of storage actors where data was not retrieved from the Voldemort store. The delaying factors which resided was the hop from one machine to another and the time taken for Voldemort to fulfill the request. Figure 6.3, reveals a similarity between the storage and service components. Both components displayed a similar decrease in response time. Given a service that has a high throughput such as Voldemort, we can expect a similar decrease in response time for requests on an external application service.

	Response time (ms)				
Number of Routees	1	10	100	1000	10000
Voldemort Store	1272	255	171	112	110
Storage	222	161	136	100	103
Service	403	315	279	226	233
Messaging	4076	4126	3592	3881	4064

Table 6.3: Average response time for the network gaming solution services against varying routee numbers in each service.

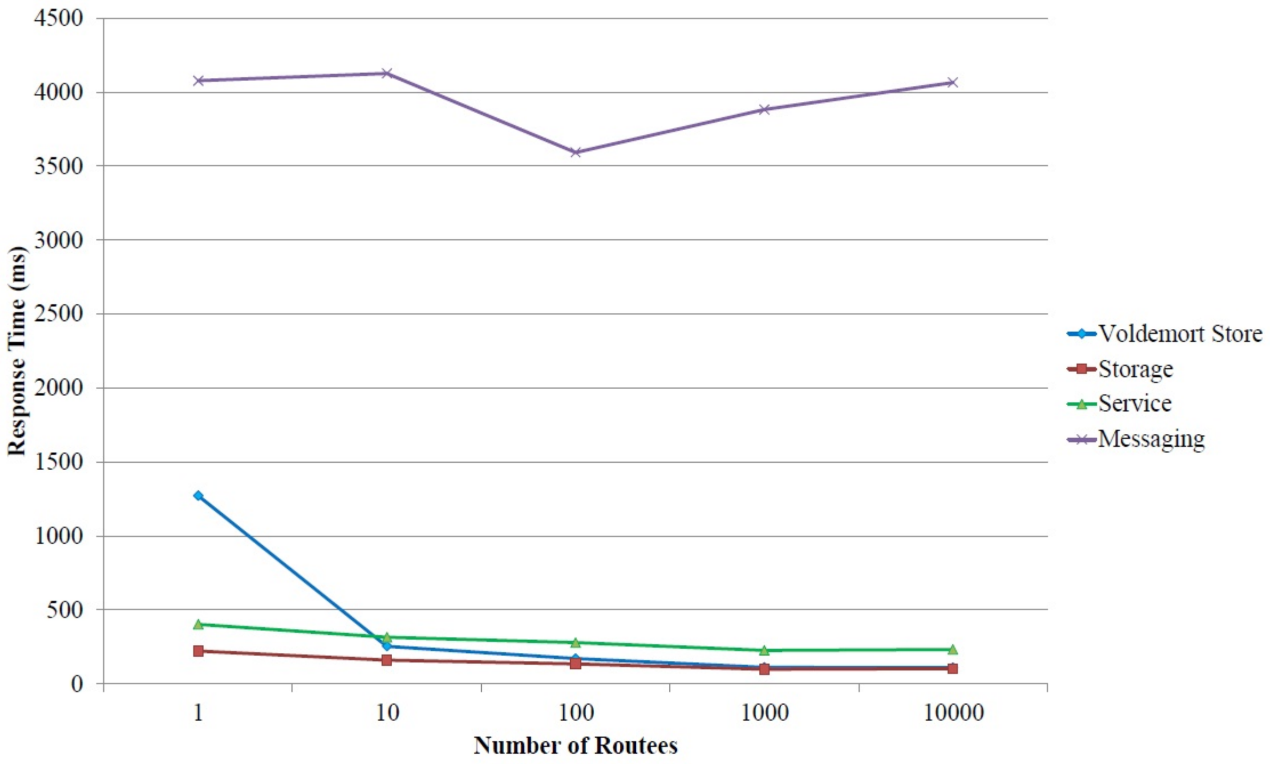


Figure 6.3: Response time comparison for the network gaming solution services against varying routee numbers in each service.

With regards to the messaging service response times, the service did not exhibit any significant improvement as the number of routees were increased. As discussed in Section 5.5, there is a two-step process for a host (which is not yet subscribed to a topic) to broadcast a message. First, the host has to subscribe to the topic by requesting subscription from the SubscriptionManagerActor. Secondly, the host sends a message to the BroadcastManagerActor, which handles the broadcast request. In order for the SubscriptionManagerActor and the BroadcastManagerActor to fulfil these requests, they both need to request the mapping of the WebSocketBroadcaster from the TopicManagerActor, making the TopicManagerActor a possible bottle neck. As shown in Figure 6.4, the messaging service can have multiple requests for subscription or broadcast during another host's request phase. For example, a host's subscription request may be in line with other hosts broadcasting requests. The TopicManagerActor manager actor is a sole actor, since its state cannot be shared unless locks are used. Consequently, the TopicManagerActor can be bombarded with a line of requests. Another factor could be that Socko's WebSocketBroadcaster is not suitable for broadcasting messages to a large number of hosts. In Section 6.5, we examine the performance of each actor in our network gaming solution to determine bottlenecks and identify where we can improve the performance of the system.

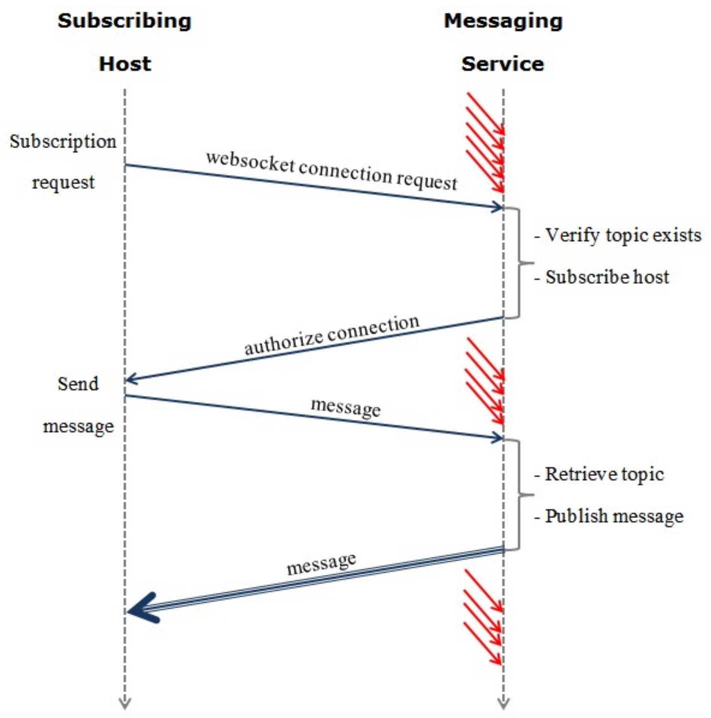


Figure 6.4: Messaging state diagram. The shorter arrows represent other hosts performing subscription and broadcasting requests.

6.4.3 Varying User Loads

In a MMOG, the amount of users playing from one set time period to another varies. For example, the change in the number of players can be attributed to the games rise in popularity. This experiment examined the response time of the system under varying user loads, indicating how the system performs as more users participate in a game. That is, if more users utilize the network, does it dramatically affect the performance of the system or can the system cope well with increasing user loads? A system with one actor per router and 100 actors per router were used to compare the response time difference. Users were simulated with JMeter and the test was conducted for times to get obtain the average response time.

Results and Analysis

Table 6.4 shows the results for the response times during user increase. Figure 6.5 displays the graphical representation of the results. Apart from confirming that increasing the number of actors reduces the response time, as in the previous section, the table also shows where a components response time starts increasing as more users are introduced. This can be used to identify where the response time becomes unsatisfactory in a single node and a distributed setting would be preferred. For example, in the messaging component, the response time is initially similar to the other components response time. Depending on the number of routee actors, in this case 100, we note that there is a large jump in response time from 200 users to 300 users. It is also evident that the response time was minimal until the number of users exceeded the number of routee actors. This displays a near perfect producer-consumer situation, where each actor serves a request from a single user.

Number of Users	Storage		Voldemort		Service		Messaging	
	1 Actor	100 Actors	1 Actor	100 Actors	1 Actor	100 Actors	1 Actor	100 Actors
10	3	5	6	5	6	6	7	8
20	4	4	6	5	6	6	8	8
30	5	4	6	6	7	7	13	8
40	6	7	7	7	7	6	18	10
50	6	5	8	7	7	6	19	11
60	8	5	10	7	7	7	17	11
70	9	5	12	8	8	7	23	18
80	9	5	15	9	9	10	24	25
90	9	5	36	9	10	9	25	30
100	8	6	79	9	18	9	53	27
200	13	9	94	10	23	13	90	26
300	15	11	104	14	24	15	129	75
400	17	11	151	16	20	25	163	137
500	20	18	209	19	22	29	198	198
600	20	20	239	21	27	31	341	261
700	26	24	308	25	29	34	421	299
800	27	25	357	23	35	32	616	316
900	31	23	371	31	40	36	827	695
1000	43	35	432	35	53	46	1093	741
2000	56	42	828	63	97	63	1086	1021
3000	65	53	956	53	102	56	985	1036
4000	90	81	1034	95	145	98	1114	1148
5000	137	106	998	153	194	153	1427	1307

Table 6.4: Average response time of the services (in milliseconds) in the peer-to-peer network solution during varying user loads.

6.4.4 Actor Fault-Tolerance

This experiment examined how an actor's failure affects the message response time. An actor restarting is a sign that the actor has experienced some sort of error. The experiment was performed in a similar manner to the test in Section 6.4.2, except specific actors were sent messages that would cause an actor to fail and restart. A Node.js application was created that was connected to a specified component and sent a message that caused an actor to fail. The interval of sending the failure message was every second. In the storage component, the StorageCoordinator was targeted to experiencing failure. Originally, the GetActor routee was set to experience failure but during preliminary tests no difference in response time was shown for the GetActor experiencing failure versus that of a GetActor operating normally. This was due to the GetActor being the bottom actor in the storage component hierarchy. If the GetActor routee experiences failure, only a single routee is affected due to the one-for-one strategy taken. By allowing the top actor in the hierarchy to fail, consequently causing all the StorageCoordinator children to fail, the results would show the response time difference during an entire components failure. In the service routing component and the messaging component, the ClientWebsocketLogActor and the TopicManageractor respectively were selected for failure. Given that these actors are not part of routers, we wished to examine how a sole actor's failure affects the performance of the system. The actors were chosen on the potential of being a bottleneck. As in Section 6.4.2, 25,000 users were simulated by JMeter and the experiment was conducted four times to retrieve the average response time.

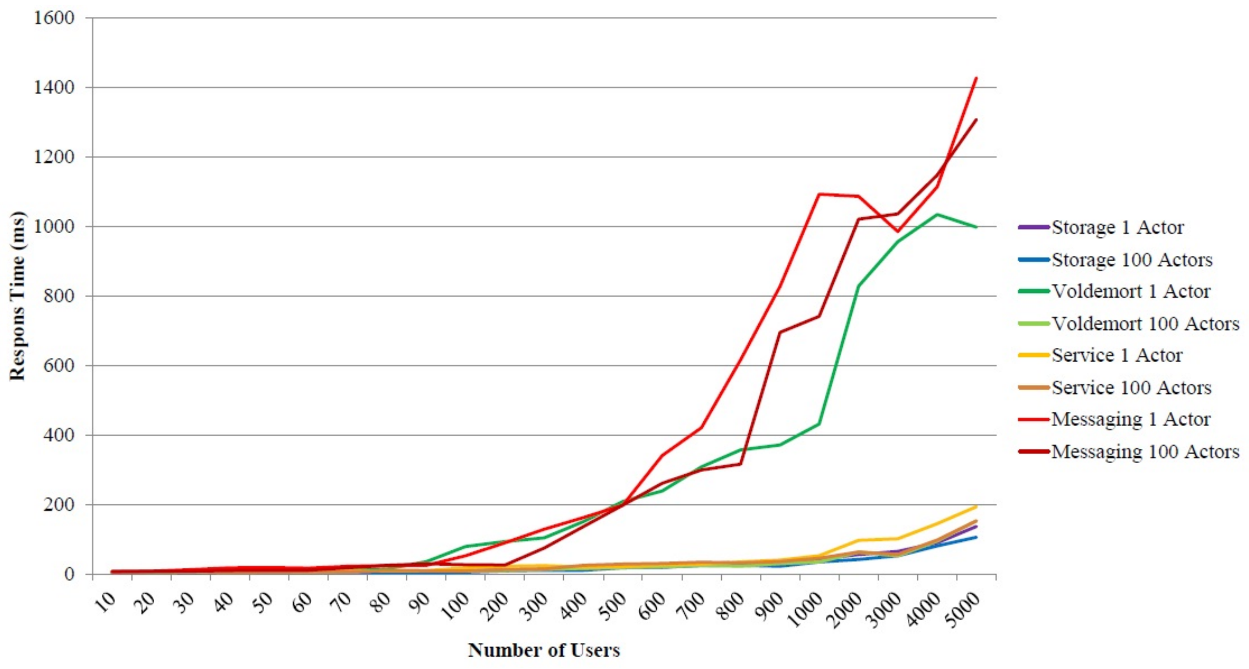


Figure 6.5: Response time comparison for the network gaming solution services against varying user loads.

Results and Analysis

Table 6.5 shows the response time during actor failure, and Figure 6.6 graphs the results. The table also lists the deviation compared to the results in Table 6.3. The storage service results (including the Voldemort store service) shows an increase in delay regarding the response time compared to the storage services not experiencing any failure in Table 6.3. The deviation increase stopped after a 1,000 routees per actor router. The delay increase was caused by the top level actor, the StorageCoordinator actor. When the StorageCoordinator actor restarts, because of failure, the child actors also have to restart. Given that a number of actors were restarting, the time taken to process a batch of messages increased. Similarly, the service routing component results exhibited the same response delay increase as more routees were employed. Though the reason for the deviation in this case is different from the one mentioned above. Given that the ClientWebsocketLogActor processes messages from two different Akka routers, the ClientWebsocketLogActor can be easily bombarded with messages from the routees. Routees are producing more messages than the ClientWebsocketLogActor can consume. The ClientWebsocketLogActor was target to be the only actor in the service routing component to be restarted, the consistent restarts caused further delay in processing the messages. The messaging module, unlike the storage and service routing modules, did not display a common deviation as routees were increased. This could be due to the bottleneck issue noted in Section 6.4.2.

Number of Routees	1		10		100		1000		10000	
	Time	Deviation	Time	Deviation	Time	Deviation	Time	Deviation	Time	Deviation
Voldemort Store	1322	-3.93%	273	-7.05%	200	-16.96%	140	-25%	122	-10.90%
Storage	230	-3.60%	173	-7.45%	132	2.94%	120	-20%	105	-1.94%
Service	410	-1.74%	360	-14.29%	351	-25.81%	360	-59.29%	323	-38.63%
Messaging	4786	-17.42%	4348	-5.38%	4641	-29.20%	4989	-28.55%	4457	-9.67%

Table 6.5: Average response time (in milliseconds) of the gaming network services during actor failure.

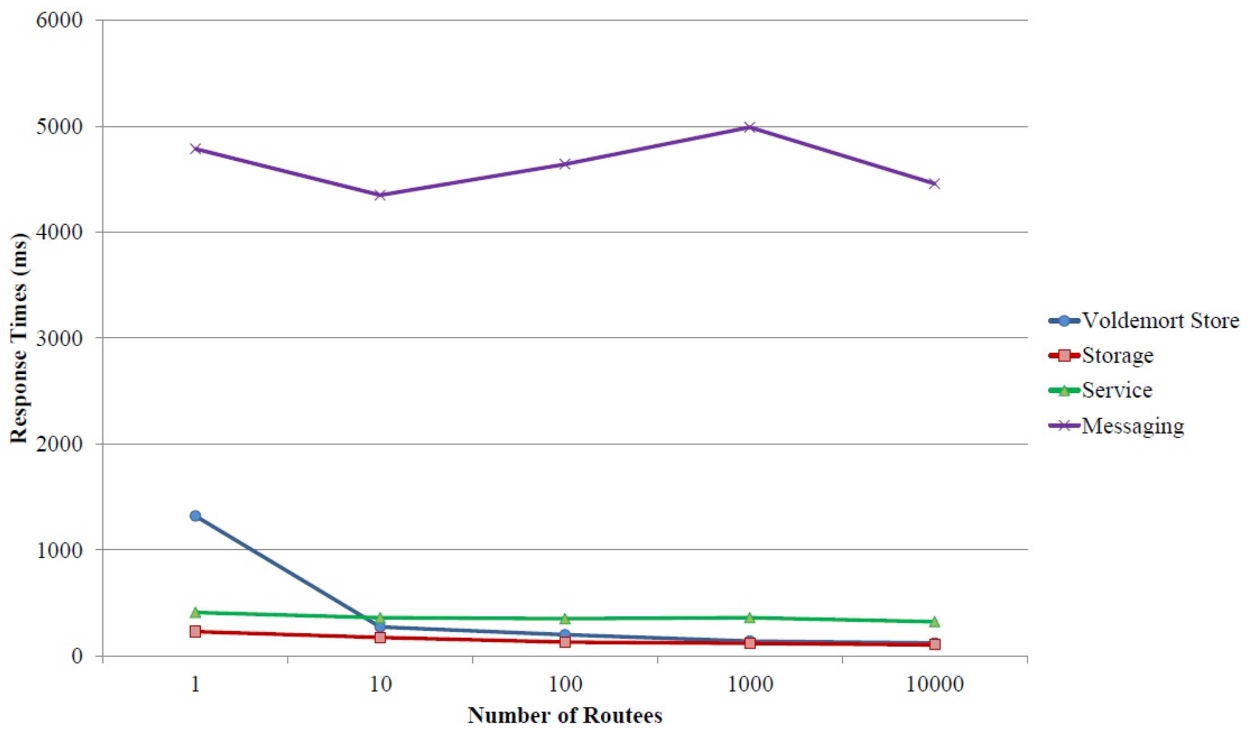


Figure 6.6: Response time graph for the network gaming solution services during actor failure.

6.5 Actor Reactive Monitoring

In Section 6.4.2, it was stated that delayed response for publishing messages in the messaging component may be due to bottlenecks in our peer-to-peer gaming solution. In this section, further examination is done on the peer-to-peer gaming solution by using the Typesafe Console to identify bottlenecks and analyse where the system can be optimized.

6.5.1 Typesafe Console

The Typesafe Console is a low overhead, transparent, real-time tracing and monitoring tool for Typesafe systems such as Akka and the Play framework [Gupta 2012]. The Typesafe Console captures events generated from an actor system. The events are linked together into a meaningful trace flow across actors (including those in remote nodes). The Console provides insight into usage trends and performance characteristics of the running system through a web browser-based interface. For Akka, the Typesafe Console can monitor different nodes (JVM), actor systems, and individual or grouped actors.

6.5.2 Evaluation Methodology

This experiment examined the performance trends of the actors in our peer-to-peer gaming solution to determine bottlenecks within the system. The Typesafe Console was used to monitor the actor performance trends. Similar to Section 6.4, the Apache JMeter tool was used to simulate user requests. A thousand users were simulated, with each user making a request four times. This experiment was conducted four times to obtain an average response time. The services provided by the peer-to-peer gaming solution were setup in the same manner discussed in Section 6.4, but with only a single actor performing a specific task, in other words, no routees were used.

6.5.3 Environment

The Voldemort node used in this experiment was the same node from the experiment conducted in Section 6.4. Only a single Apache JMeter node was used containing the specifications from the experiment in Section 6.4. Below, we list the specifications of the machine that hosted the peer-to-peer gaming solution.

Gaming solution node:

- Intel Core 2 Duo T5800 @ 2.00GHz
- 2 GB RAM
- 300 GB available hard drive storage capacity
- Ubuntu 12.04 (precise) 32-bit
- Kernel Linux 3.2.0-29-generic
- File descriptor: 512 000 for soft value; 1 024 000 for hard value
- Akka 2.2.0
- Socko 0.3.1
- Java 1.7.0_10 (1 GB heap memory)
- Typesafe Console 1.3.0

6.5.4 Results and Analyses

Tables 6.6, 6.7, 6.8 and 6.9 respectively provide usage trends of the storage module, the storage module retrieving values from a Voldemort store, the messaging module and the service routing module. The performance characteristics of the actors affected by the user requests, as well as the entire actor system, are listed in the tables. The following attributes were monitored during the tests:

- Throughput (message rate): The rate at which messages are processed in a single second.
- Mailbox Size: The number of messages in a mailbox.
- Mailbox Time: The amount of time (in milliseconds) a message spends in a mailbox waiting to be processed by an actor.
- Latency: Indicates the time duration (in milliseconds) from when a message was sent to an actor until the message processing has been completed by the receiving actor.

	Total Throughput (msg/sec)	Mean Throughput (msg/sec)	Max Mailbox Size	Max time in Mailbox (ms)	Mean Time in Mailbox (ms)	Mean Latency (ms)
StorageCoordinator	611.1	18.36	909	3284	0.35	1803
GetActor	618.7	18.27	70	90.63	0.18	9.252
System	926.3	40.18	909	3284	820.7	819.8

Table 6.6: Performance characteristics of the storage component actors.

Comparing Table 6.6 and 6.7, the results reflect those obtained in Section 6.4. When retrieving values from the Voldemort store, throughput decreased and latency increased accordingly. By increasing the

	Total Throughput (msg/sec)	Mean Throughput (msg/sec)	Max Mailbox Size	Max time in Mailbox (ms)	Mean Time in Mailbox (ms)	Mean Latency (ms)
StorageCoordinator	410.7	22.21	821	5592	4	1767
GetActor	457.15	27.77	810	5011.5	2	1760.5
System	565.6	60.99	821	5592	1525.5	1726

Table 6.7: Performance characteristics of the storage component actors retrieving values from the Voldemort store.

number of actors, as in Section 6.4.2, throughput should increase and latency should decrease. Table 6.8 indicates that the ClientRequestServiceActor and the ServiceDispatcherActor are the slowest actors in the service module. Both actors had a fairly high latency compared to the other actors in the service routing module. The ClientRequestServiceActor also had the maximum mailbox waiting time. Concerning the actor hierarchy in the service routing component, both actors are the top children in the service component. The delay in message processing is likely due to the JSON deserialization the actors perform. Given that several instances of the ClientRequestServiceActor and the ServiceDispatcherActor can be created, distributing the JSON deserialization amongst sever actors would decrease the latency experienced by the actors.

In Section 6.4.2, we assumed that the TopicManagerActor was the bottleneck in the messaging component. Table 6.9 shows that the Socko WebSocketBroadcaster was responsible for the systems considerably large time span in broadcasting a message. The table indicates that at some point a message waited over 21 seconds in the WebSocketBroadcaster's mailbox before it was processed. More importantly, the average mean latency of the WebSocketBroadcaster was higher than any other actor observed during the tests.

	Total Throughput (msg/sec)	Mean Throughput (msg/sec)	Max Mailbox Size	Max time in Mailbox (ms)	Mean Time in Mailbox (ms)	Mean Latency (ms)
ClientRequestServiceActor	134.1	18	531	2500	1.0	993.4
ClientWebsocketLogActor	540.9	35.7	612	706.4	0.13	23.8
ClientRequestRouter	99.59	10.41	164	685.3	0.09	14.87
ServiceDispatcherActor	651.3	63.22	6704	1824	0.947	1083
ResponseDecoderActor	111.8	10.62	103	106.6	0.005	7.921
ServiceResponseActor	192.2	12.22	100	108.3	0.009	8.825
System	2026	155	6704	2500	490.4	547.5

Table 6.8: Performance characteristics of the service routing component actors.

	Total Throughput (msg/sec)	Mean Throughput (msg/sec)	Max Mailbox Size	Max time in Mailbox (ms)	Mean Time in Mailbox (ms)	Mean Latency (ms)
SubscriptionManagerActor	16.63	4.766	59	250.7	0.07	55.86
BroadcastManagerActor	44.81	11.31	406	1902	0.752	153.5
TopicManagerActor	44.81	16	52	153.9	0.39	7.492
WebSocketBroadcaster	35.94	21.95	2830	21940	29.9	14560
System	63.82	87.12	2830	21940	5999	6247

Table 6.9: Performance characteristics of the messaging component actors.

6.6 Distributed Environment Evaluation

The experiment examined the response time of the gaming network solution in a distributed environment to demonstrate that distributing a game server's functions (such as message dissemination and state management) reduces the response time to high user loads and allows more players to participate in the game. The experiment gave emulated the peer-to-peer gaming solution we aimed to provide. The experiment was conducted by placing our gaming network on several nodes. The test focused on the response times of the services offered by the gaming network solution. The Nginx reverse proxy was used to distribute requests amongst the nodes. The supernode system which we developed requires that a specific backend node be given for each request. That is, we have to manually set a group of users to make requests from a specific node. The Nginx reverse proxy evenly distributes requests without specifying the backend node. Figure 6.7 shows a graphical representation of the test setup. The experiment also observed the retrieval time of values from a distributed Voldemort storage. The comparison between using a single storage node and multiple storage nodes gave us insight into the retrieval time of game data in a distributed setting.

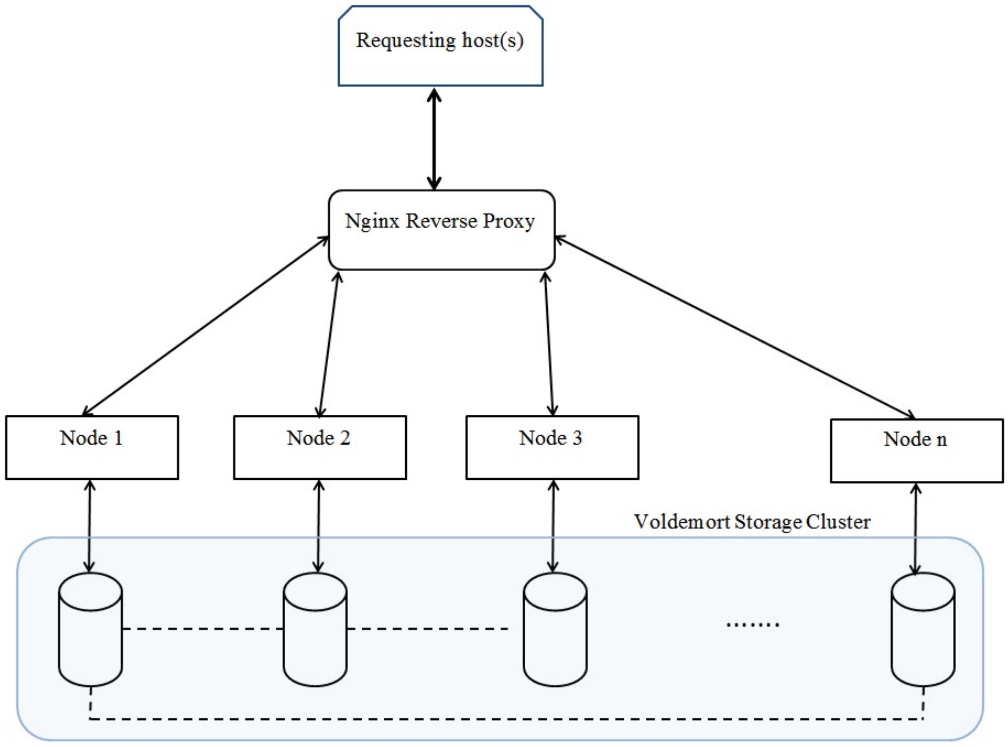


Figure 6.7: Graphical representation of the multiple node setup.

6.6.1 Evaluation Methodology

In this experiment, we examined the response time from the services provided by our gaming network solution in a distributed environment. These services were the storage function, the message dissemination function, and the service routing function. As in Section 6.4, we examined the storage in two forms: (i) when data is retrieved from the Voldemort storage, and (ii) when response values are generated by the GetActor. Furthermore, we extend the number of Voldemort storage nodes, distributing the data across several nodes. 200,000 values were generated and spread across the Voldemort storage cluster nodes. The messaging and service routing components were configured in the same manner as in Section 6.4. To ensure the response time comparison is only factored by the distribution of the gaming network across several nodes, services containing router actors had a single route performing specific tasks. That is, multiple routees were not used in this experiment. 25,000 users were simulated through JMeter, with each user making four requests. The tests were conducted four times to remove the presence of unforeseen network activity.

6.6.2 Environment

Below, we list the specifications of the machines which were used during this experiment.

The Hydra Cluster

The Hydra cluster is a cluster of machines located at The University of the Witwatersrand. Five nodes from the Hydra cluster were allocated to host the peer-to-peer gaming network. Each node in the Hydra cluster had the following setup:

- Intel Core i7 CPU 3.0 GHz
- 6 GB RAM
- 200 GB hard drive storage capacity
- Ubuntu 12.04 (precise) 64-bit
- Kernel Linux 3.8.0-29-generic
- Akka 2.2.0
- Socko 0.3.1
- Java 1.7.0_10 (5 GB heap memory)
- File descriptor: 512 000 for soft value; 1 024 000 for hard value

Voldemort Nodes

Five nodes from the Hydra cluster were allocated for the Voldemort storage. Each Voldemort storage node used 5 GB of JVM memory heap. Following the evaluation model of Voldemort conducted by Sumbaly *et al.* [2012], the replication factor was set to 1. Sumbaly *et al.* [2012] states that latency is a function of the data size and should be independent of the replication factor. Each Voldemort node had the following configurations:

- Voldemort version 1.3.0
- Persistence storage engine: BDB
- Routing policy: client
- Replication factor: 1
- Required read and writes: 1
- Number of partitions: 1

Apache JMeter

The Apache Jmeter setup was the same as that in Section 6.4.1.

Nginx Reverse Proxy Node

The Nginx reverse proxy node contained the following specifications:

- Intel Core i7 @ 3.40 GHz - 3770
- 8 GB RAM
- 500 GB hard drive storage capacity
- Ubuntu 12.04 (precise) 64-bit
- Kernel Linux 3.8.0-29-generic
- File descriptor: 512 000 for soft value; 1 024 000 for hard value
- Version: 1.5.6
- Worker processes: 16
- Worker connections: 400 000
- Maximum file descriptors (worker_rlimit_nofile): 1 024 000

6.6.3 Results and Analysis

Table 6.10 and Figure 6.8 show the response time from a single node system versus a multi-node system. The results indicate that by introducing more nodes, the time taken for a response decreases. Work load was distributed amongst the nodes equally which allowed requests to be completed faster than with a single node handling all the requests. More importantly, the messaging service exhibited a huge decrease in response time, something which was unachievable by just increasing the router actors in Section 6.4.2. By distributing the message dissemination function amongst the five nodes, the distributed network achieved a 79% reduction in response time. However, the response time in the message dissemination service is still fairly high compared to the other services offered. Therefore, the bottleneck attributed to Socko's WebSocketBroadcaster is still prevalent. We also note an increasing delay for a response from the Voldemort storage cluster when using a single storage node (208 milliseconds) versus that of multiple storage nodes (388 milliseconds) in the distributed network results. The increased delay is likely due to the Voldemort client in our system computing the location and searching for the value amongst the Voldemort nodes. Replicating the data across several nodes is likely to decrease the time a Voldemort client spends searching for a value in the cluster. In summary, the results obtained from this experiment indicate that a distributed environment decreases the time taken to process network gaming messages, particularly in the case where there is a large user base. Furthermore, we can use the results obtained from Section 6.4.3 to determine when it is suitable to distribute a game servers functions amongst the peers in the network.

			Rate of Improvement
Number of Nodes	1	5	
Storage Service	145	87	40%
1 Voldemort Storage Node	1651	208	87%
5 Voldemort Storage Nodes	1406	388	72%
Routing Service	207	116	44%
Messaging Service	4603	975	79%

Table 6.10: Average response time comparison (in milliseconds) of the gaming network solution in a single node environment versus a distributed environment.

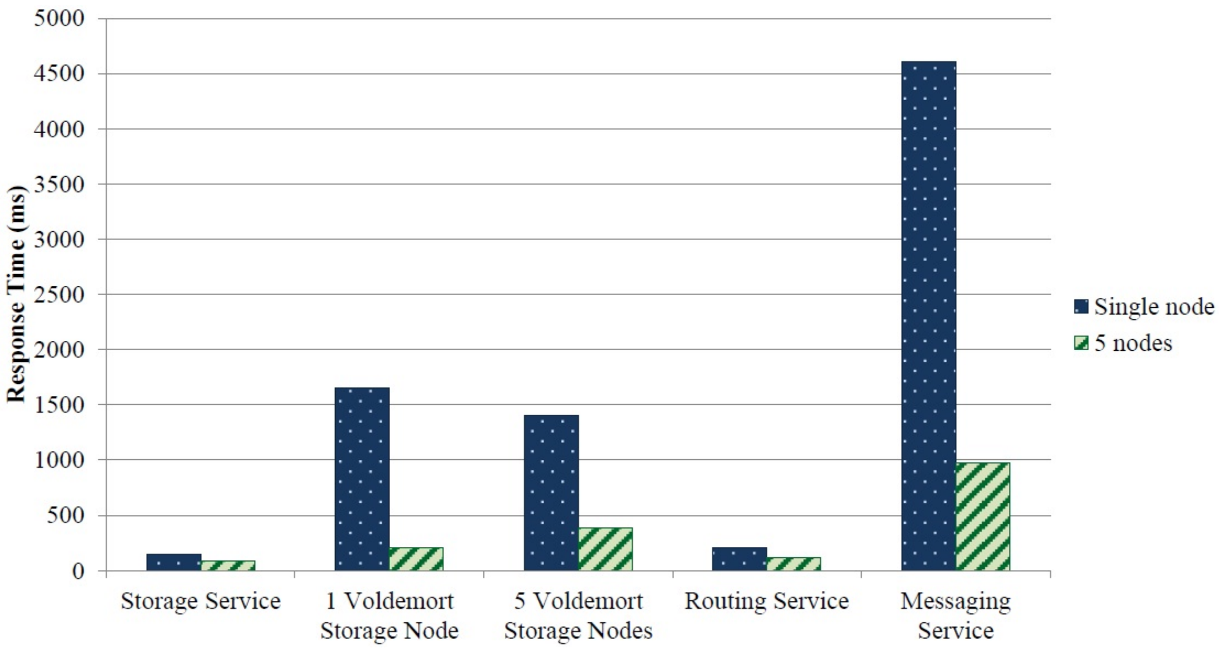


Figure 6.8: Graphical response time comparison of the gaming network solution in a single node environment versus a distributed environment.

6.7 Conclusion

This Chapter focused on presenting the results used to evaluate our peer-to-peer gaming network. We wished to provide a low latency and reliable communication mechanism for peer-to-peer MMOG. Despite TCP being a reliable transport protocol, TCP's packet order guarantee property causes delays that cannot be tolerated by certain games. With the Transmission Control Protocol, when a packet is dropped by the network, an application cannot receive further messages from the network until the missing packet has been retransmitted. By using SPDY's multiplexing stream attribute, it was shown that the delay factor of applications using TCP can be mitigated. SPDY's multiplexing property allows for network applications to continue receiving messages, in the event of packet loss. Furthermore, SPDY displayed an improvement on data transfer time when compared to HTTP. We also noted that SPDY's connection setup could be a problem for online games operating on networks with low bandwidth. However, we stated in Chapter 1 that the increasing interest in peer-to-peer network applications is due to the improvement of home bandwidth. Therefore, the majority of home networks have adequate bandwidth to mitigate SPDY's connection overhead.

The chapter also examined the performance of our peer-to-peer solution. The event-driven paradigm provided high availability. In addition, the Akka toolkit provided high throughput and low latency in processing network messages. By increasing the number of routees, for a specific instance of an actor, the time taken to process messages was decreased. Given Akka's fault-tolerance feature, we also examined the delay a fault can cause to operation of an online game. The results showed that faults that constantly occur will delay the processing of messages. The delay factor was acceptable for some of the system functions, which was dependent on which actor we selected to fail. Additionally, the option of a fault occurring and being able to automatically restart specific service is more advantageous than the entire system failing, requiring intervention by the user. With regards to the services provided by our peer-to-peer gaming solution, we noted that the message dissemination function is unable to handle high user loads, but performs adequately on a small user scale. Further investigation revealed that Socko's WebSocketBroadcaster was responsible for the delay in broadcasting messages in the message dissemination function. We discuss solving this issue in Chapter 7.

To provide a comparison on our systems performance to other online gaming solutions, a benchmark was performed comparing our solutions front-end service against other web servers with online gaming capabilities. The web servers were Nginx, Apache HTTP and Node.js. The benchmark compared the time taken to serve a web page to a varying number of users. Results showed that Socko performed slightly better than Apache HTTP, currently the most popular web server, and outperformed Node.js as the number of users increased. On the other hand, Nginx displayed resilient performance against the other servers, even as the number of users were increased. Overall, Socko provides adequate performance for hosting online multiplayer games. Tuning the Socko and Akka configurations will likely produce better performance for online multiplayer games.

Finally, we examined our system in a distributed setting. One of the main desires in using a peer-to-peer system for online games was to distribute the computational work and hardware requirements for hosting an online game. By distributing the workload, we wished to increase the number of players that can participate in an online game, a factor which is difficult and highly expensive to achieve in a client-server architecture. The experiment compared the response time against multiple nodes in a distributed environment versus that of a single node. The tests showed that the response time for serving requests greatly improved in a distributed environment compared to a client-server setting. The improvement shows that our system is able to scale as more nodes are added, achieving the goal of scalability set in Chapter 3.

A reductionist approach was taken to evaluate the peer-to-peer gaming network developed. Rather than evaluating the peer-to-peer infrastructure on a network game, discrete elements were identified and evaluated in order to answer the research questions. Due to time constraints, we were unable to investigate the performance of the peer-to-peer system as a whole, and not just the specific elements given in this chapter. Except for the experiment presented in Section 6.2, the experiments presented in this dissertation were conducted across a local area network. This exposes an issue for MMOGs as nodes are likely to be distributed widely over a network. Investigation needs to be conducted on the performance of the peer-to-peer solution for nodes located in different physical regions, or by simulating a network which would appear to be connected over different locations.

Chapter 7

Conclusions, Contributions and Future Work

7.1 Conclusion

The growth and popularity of Massive Multiplayer Online Games (MMOG) has introduced a range of interesting challenges. MMOG use the client-server model to provide the online gaming infrastructure. Client-server architectures do not scale well as more hosts use the network and are expensive to maintain. Also, the server becomes the single point of failure for any application using the client-server architecture. Given the exponential growth of the number of video games with online playability, there has been a collective interest on the design of hosting online games over peer-to-peer networks. A peer-to-peer architecture has the ability to reduce the cost of maintaining servers, increase the scalability of games and provide fault-tolerance against a host failing. Though peer-to-peer architectures aim to resolve the problems with client-server architectures, peer-to-peer architectures also yield some challenges that must be addressed. Namely: game state management, distributed storage, message dissemination model, optimization of delays, player synchronisation and protection against cheating.

This research focused on the design and implementation of a peer-to-peer architecture that would reliably deliver game messages in a timely manner. Games may experience delays in two forms: the network delay, the time taken to transport a message over the network, and synchronisation delay, the time taken to account for all the players' actions and synchronise the players state. In online games, the delay must be kept to a minimum to avoid lag. Most of the techniques used to minimize synchronisation delays have been developed over the years for client-server game models, and can be adapted to peer-to-peer architectures. Network delays are attributed to the transport layer and application layer protocol. In context of the transport layer, UDP is often used to transport game messages across a network in action-based games. TCP has been noted to cause delays for a number of games due to its property of reliably delivering messages. UDP's drawback was that it does not guarantee the delivery of messages and has no flow control (congestion avoidance). Also, TCP can easily traverse networks with firewalls, proxies and authentication servers. By using the application layer protocol SPDY, we showed that it is possible to deliver messages over the Transmission Control Protocol in a timely manner.

The peer-to-peer gaming network we developed provides three functions: message dissemination, service routing and distributed data storage. The peer-to-peer network application was built on top of the event-driven actor paradigm. Unlike thread-based systems, actors do not require locks or monitors to share data. Consequently, actors are likely to process tasks faster than thread-based paradigms. The experiments conducted in the previous chapter showed that by increasing the number of actors to perform tasks related to the data storage and service routing functionalities, thereby distributing the workload, resulted in a decrease of the time taken to process a message. However, increasing the number of actors in the message dissemination component did not provide any positive results. Through further investigation, it was discovered the message dissemination component contained a bottleneck. The bottleneck was caused by an actor in the Socko framework which publishes messages to hosts subscribed to a topic.

The research also revealed a relation between game state management and distributed data systems. Modern non-relational database management systems, known as NoSQL systems, provide a weakly consistent distributed storage system in exchange for availability and partition-tolerance. Video games tolerate weak consistency of the game state. Most of the read and write operations occur in the players Area of Interest (AOI), therefore game servers are able to present a globally weak consistent state. From this, a low latency distributed storage system, Voldemort, was employed in the peer-to-peer gaming solution. The intent of the Voldemort storage system was to store game state data and provide fault-tolerance against a node abruptly leaving the peer-to-peer system. Also, Voldemort's eventual consistency property allows the system to keep a regions state consistent and gradually propagate the state to other regions.

The peer-to-peer gaming solution developed was successfully examined on the University of the Witwatersrand's Hydra cluster. The test aimed to simulate the case where a game world is partitioned into regions and each node manages a region of the game world. The investigation revealed that distributing the game network solution favourably decreased the time to receive a response from the network functionalities. The distributed data storage, service routing and message dissemination functionalities all exhibit a decrease in response latency.

Reviewing the research aims discussed in Chapter 3, we proposed a peer-to-peer network system for hosting online multiplayer games as alternative to client-server architectures. This research presented a peer-to-peer MMOG architecture with the following functionalities:

- Consistency and Availability
  In MMOG, players' game state must be consistent while allowing all read write request to be fulfilled. Relating to the CAP theorem introduced in Chapter 4, we adopted the notion of eventual consistency by guaranteeing local consistency and weak global consistency. Eventual consistency provides high availability and partition-tolerance. The peer-to-peer gaming system was designed around the Supernode Control protocol in order to acquire a dominant node that would maintain local consistency. The Voldemort distributed storage system was used to store game state data.

- Fault-tolerance
  In distributed systems, a node may unexpectedly fail which in turn may hinder the game playability. Voldemort automatically replicates game state data over multiple nodes. If a node were to abruptly disconnect from the network, the game state of the failed node can be retrieved from the copies contained on several nodes. The Akka actor toolkit also provided resistance against faults. When a component in the system fails, Akka is able to restart the component back into a stable state, allowing for game play to continue.

- Scalability
  The architecture is designed to handle a large number of concurrent players. By distributing the architecture functions such as message dissemination and game state management, the architecture can support a large number of players compared to client-server architectures. Also, Akka's load distribution feature allows for more users to participate on a single host.

- Low latency
  Many multiplayer online games require that messages arrive under a certain time to avoid the user experiencing lag. SPDY was used to control how messages are sent across the network on top of TCP. SPDY was shown to reliably deliver messages and diminish the delay associated with the TCP transport protocol. The actor event-based paradigm was used to rapidly process network messages. By increasing the number of routees (actor instances that perform a specific task) the actor based system decreased the amount of time taken to process network messages.

In conclusion, the research question posed has been resolved. By using SPDY for network communication and actors to process messages, a low latency method was developed to reliably deliver messages for peer-to-peer games.

7.2 Contributions and Future work

This work contributed a new extension to online gaming networks. Deciding on the transport layer is dependent on the type of game. UDP is preferred over TCP in action-based games due to the cost of TCP's order guarantee property. In the case of TCP, when packets arrive at the receiving host, the data is only passed onto the application-layer only if previous packets have arrived. If a packet was lost during the transition from the sending host to the receiving host, other packets will be held in a buffer. These buffered packets will only be passed onto the application-layer once the lost packet has been retransmitted. With UDP, any data which is sent to the receiving host is immediately pushed onto the application layer. In this research, it was shown that SPDY's multiplexing property is able alleviate the issue faced with TCP's order arrival property. This observation is likely to open up further discussions concerning the transport protocol and online gaming networks. Though the peer-to-peer solution we developed is aimed at MMOG, the peer-to-peer network can extend beyond online gaming. Any application that seeks a similar networking infrastructure that we developed may likely use our peer-to-peer network application. A set of attributes in our network application make it easy to adopt for applications beyond gaming; including the use of JSON, a platform independent serialization tool, and running the network application on JVM, allowing for portability to different systems.

The research study tackled certain aspects facing peer-to-peer gaming networks. But many challenges still exist that were not addressed fully. A complete peer-to-peer gaming solution needs to be developed. For example, issues surrounding cheating were not resolved. Also, protocols for managing game world regions and determining when a region should be partitioned further need to be integrated into our peer-to-peer gaming solution.

Throughout this dissertation, we noted issues that need to be addressed in the future. In particular:

- The bottleneck within the message dissemination function should be addressed.
- The connection between the network gaming application and Voldemort was noted to be a standard TCP connection. A SPDY connection between the two systems should be implemented.
- In our networking solution, only a few actors state are persistent. All the actors states must be persisted to ensure message delivery guarantee amongst the actors.

In this research, we evaluated the TCP network delay against SPDY and the HTTP communication protocol. Given that there are several UDP gaming communication protocols [Harcsik *et al.* 2007], each with different performance statistics dependent on the type of game, we did not conduct any analysis against these protocols due to time constraints. In future, we wish to examine the performance of SPDY against other online gaming communication protocols. Prioritizing messages in the actor system is another improvement that can reduce the latency. Certain games may require that specific messages have a priority in being processed by the network application [Aldridge 2011]. For example, in a first person shooter game, in order for the hit detection algorithm to accurately calculate if a player was hit or not, the message indicating the shooting needs to processed as soon as possible. SPDY already has a messaging prioritization property. Messages need to be prioritized on the actor side of our peer-to-peer gaming solution. By assigning message priority, a message can be put in front of the queue of an actor's mailbox, ensuring it gets processed next.

Another task to pursue in the future is to ensure that peer-to-peer solution developed is suitable for product use. Currently, to add node to the Voldemort storage cluster, while the cluster is operational, requires a number of steps to be completed by the user. An easy and straightforward interface needs to be developed to add/remove nodes from the Voldemort cluster. Applying the peer-to-peer solution on a game and assessing the users gaming pleasure is the final benchmark in proving that peer-to-peer gaming solutions are a viable option in solving issues facing current MMOG technology.

Recently, game developers have adopted the concept of content on demand through cloud-based game streaming services. Cloud gaming, in its simplest form renders interactive games in the cloud and streams the scenes as a video back to the player [Shea *et al.* 2013]. Players interact with the game through a thin client. The thin client is solely responsible for displaying the video content rendered by the cloud service, as well as collecting the player's interactions with the game and sending the actions to the cloud for rendering. Cloud computing is advantageous for less powerful computational devices (such as mobile devices) that are incapable of running resource intensive games. Cloud computing also resolves the issue of compatibility between games and devices.

Gaikai[1] and OnLive[2] are the two of the most prominent cloud-based gaming services [Shea *et al.* 2013]. Recently, Sony acquired Gaikai in plans to implement a game streaming service for Sony gaming consoles (Playstation 3 and 4), televisions and smartphones [Hollister 2014]. Cloud-based games can also benefit from peer-to-peer architectures [Yahyavi and Kemme 2013]. Cloud systems based on peer-to-peer networks can resolve the problematic bottleneck often experienced in cloud computing systems [Xu *et al.* 2009]. There is also the possibility of creating a fully decentralized cloud-based gaming network [Babaoglu *et al.* 2012].

[1]www.gaikai.com
[2]www.onlive.com

Appendix A

API

The application programming interface (API) used to interact with the functions in the peer-to-peer gamin solution is presented below. Table A.1, A.2 and A.3 respectively contain the API for the storage component, message dissemination component and service routing component. The message dissemination API is contains two interfaces. The first is used to create and delete topics for hosts to broadcast messages through. The second is used to publish messages to a specific topic. The service routing API is also separated into two interfaces. The *Service* interface is used by an external application to register its service and respond to a client requests. The *Service Client* interface is used by clients to send requests to a service.

Storage			
URL	/storage		
sendMessage(operation : String, storeName : String, key : Object, value : Object)			
	operation	delete	Delete any version of the given key which equal to or less than the current versions
		get	Get the value associated with the given key
		put	Associates the given value to the key, clobbering any existing values stored for the key
responseMessage(status : String, operation : String, storeName : String, key : Object, value : Object)			
	status	succes	Operation was successful
		failure	Operation failed
	operation	getResponse	The message contains a response from the get operation

Table A.1: The API for the storage component

Messaging			
Topic Mangement			
URL	/topicmanagement		
topicOperation(topicName : String, task : String)			
	task	create	create a topic for publication of messages
		remove	delete a topic for publication of messages
Message Broadcast			
URL	/messaging/*topicName*		
	Pass any message to the to the specified URL and the message will broadcasted to the subscribed hosts		

Table A.2: The API for the message dissemination component

Service Routing			
Service			
URL	/service		
serviceDispatchMessage(dispatcher : String, data : String)			
	data	serviceManagement()	JSON stringified message
		responseMessage()	JSON stringified message
serviceManagement(operation : String, workername : String)			
	operation	add	add a worker/service to the node
		delete	delete a worker/service from the node
responseMessage(worker : String, operationData : String, clientChannel : Int)			
requestMessage(worker : String, operationData : string, clientChannel : Int, response : Boolean)			
Service Client			
URL	/computation		
servicerequest(worker : String, operationData : string, response : Boolean)			
serviceResponse(worker : String, operationData : String)			

Table A.3: The API for the service routing component

Appendix B

SPDY Benchmark Results

Table B.1 presents the results from an experiment conducted by Google, comparing the performance of SPDY against HTTP [Google 2012]. The test consisted of downloading 25 of the top 100 websites over a simulated home network connection, with 1% packet loss.

	2 Mbps downlink, 375 kbps uplink		4 Mbps downlink, 1 Mbps uplink	
	Average ms	**Speedup**	**Average ms**	**Speedup**
HTTP	3111.916		2348.188	
SPDY basic multi-domain connection / TCP	2242.756	27.93%	1325.46	43.55%
SPDY basic single-domain connection / TCP	1695.72	45.51%	933.836	60.23%
SPDY single-domain + server push / TCP	1671.28	46.29%	950.764	59.51%
SPDY single-domain + server hint / TCP	1608.928	48.30%	856.356	63.53%
SPDY basic single-domain / SSL	1899.744	38.95%	1099.444	53.18%
SPDY single-domain + client prefetch / SSL	1781.864	42.74%	1047.308	55.40%

Table B.1: Average page load times for top 25 websites [Google 2012].

Tables B.2 and B.3 present results from experiments that measured if the packet loss rate and the RTT had any effect on the performance of SPDY. The experiments were conducted on a simulated cable link (4 Mbps downlink, 1 Mbps uplink). Like the experiments presented above, the tests consisted of downloading 25 of the top 100 websites, but a variance in packet loss and RTT was simulated.

	Average ms		**Speedup**
Packet loss rate	HTTP	SPDY	
0%	1152	1016	11.81%
0.50%	1638	1105	32.54%
1%	2060	1200	41.75%
1.50%	2372	1394	41.23%
2%	2904	1537	47.70%
2.50%	3028	1707	43.63%

Table B.2: Average page load times for top 25 websites by packet loss rate [Google 2012].

	Average ms		**Speedup**
RTT in ms	HTTP	SPDY	
20	1240	1087	12.34%
40	1571	1279	18.59%
60	1909	1526	20.06%
80	2268	1727	23.85%
120	2927	2240	23.47%
160	3650	2772	24.05%
200	4498	3293	26.79%

Table B.3: Average page load times for top 25 websites by RTT [Google 2012].

References

[Adya *et al.* 2002] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[Agarwal and Lorch 2009] Sharad Agarwal and Jacob R. Lorch. Matchmaking for online games and other latency-sensitive P2P systems. *SIGCOMM Comput. Commun. Rev.*, 39(4):315–326, August 2009.

[Aldridge 2011] David Aldridge. I shot you first: Networking the gameplay of HALO: REACH. In *Proceedings of the 15th Games Developers Conference*. Bungie LLC, 2011.

[Anderson *et al.* 2010] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.

[Anstey 2009] Jonathan Anstey. *Apache Camel: Integration Nirvana*. Dzone blog, 2009. Retrieved November 2013, from `http://architects.dzone.com/articles/apache-camel-integration`

[Armstrong 2007] Joe Armstrong. *What's all this fuss about Erlang?* The Pragmatic Bookshelf blog, 2007. Retrieved October 2013, from `http://pragprog.com/articles/erlang`

[Armstrong 2010] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, September 2010.

[Arts 2014a] Electronic Arts. *EA*. online, January 2014. Retrieved January 2014, from `http://www.ea.com`

[Arts 2014b] Electronic Arts. *Need for Speed*. online, January 2014. Retrieved January 2014, from `http://www.needforspeed.com/`

[Babaoglu *et al.* 2012] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. Design and implementation of a p2p cloud system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 412–417, New York, NY, USA, 2012. ACM.

[Bandai 2014] Namco Bandai. *Tekken*. online, January 2014. Retrieved January 2014, from `http://www.tekken.com/`

[Barri *et al.* 2010] Ignasi Barri, Francesc Gin, and Concepci Roig. A scalable hybrid p2p system for mmofps. In Marco Danelutto, Julien Bourgeois, and Tom Gross, editors, *PDP*, pages 341–347. IEEE Computer Society, 2010.

[Barton 2007] Matt Barton. *The History of Computer Role-Playing Games Part 1: The Early Years (1980-1983)*. Gamasutra Website, February 2007. Retrieved October 2013, from `http://www.gamasutra.com/view/feature/3623/the_history_of_computer_.php`

[Bell 1998] Joe Grant Bell. *Command and Conquer: Red Alert Retaliation*. Prima Communications, Inc., Rocklin, CA, USA, 1998.

[Belshe and Peon 2012] Mike Belshe and Roberto Peon. SPDY protocol draft-mbelshe-httpbis-spdy-00. In *Network Working Group*. Internet Engineering Task Force, 2012.

[Belshe *et al.* 2013] M. Belshe, Peon R., M. Thomson, and Melnikov A. *Hypertext Transfer Protocol version 2.0 — draft-ietf-httpbis-http2-04*, 2013.

[Ben-Ari 1990] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Berners-Lee *et al.* 1996] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, 1996.

[Bernier 2001] Yahn W. Bernier. Latency Compensating Methods in Client/Server In-Game Protocol Design and Optimization. In *Proceedings of the 15th Games Developers Conference*, March 2001.

[Bharambe *et al.* 2006] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[Biscotti *et al.* 2011] Fabrizio Biscotti, Brian Blau, John-David Lovelock, Tuong Huy Nguyen, Jon Erensen, Shalini Verma, and Venecia K Liu. Market trends: Gaming ecosystem, 2011. volume 2011, Stamford, NY, USA, 2011.

[Blizzard 2014] Blizzard. *Blizzard Entertainment*. online, January 2014. Retrieved January 2014, from `http://www.blizzard.com`

[Bonér 2011] Jonas Bonér. *Akka - an open source, event-driven middleware project*. Electronic Design Website, September 2011. Retrieved November 2013, from `http://electronicdesign.com/embedded/akka-open-source-event-driven-middleware-project`

[Boulanger *et al.* 2006] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA, 2006. ACM.

[Braden 1989] Robert Braden. *Requirements for Internet Hosts - Communication Layers*. RFC Editor, United States, 1989.

[Brewer 2000] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[Brown and Wilson 2011] Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*. lulu.com, June 2011.

[Capcom 2014] Capcom. *The World of Street Fighter*. online, January 2014. Retrieved January 2014, from `http://www.streetfighter.com/`

[Carlson 2013] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[Chang *et al.* 2006] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[Chen-Becker 2011] Derek Chen-Becker. *JSON parser*. Scala Mailing List, 2011. Retrieved July 2013, from `https://groups.google.com/d/topic/scala-user/P7-8PEUUj6A/discussion`

[Clinger 1981] William D Clinger. *Foundations of Actor Semantics*. Technical report, Cambridge, MA, USA, 1981.

[Codd 1979] Edgar F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, December 1979.

[Cohen 2003] Bram Cohen. *Incentives Build Robustness in BitTorrent*, May 2003.

[Community 2013] Netty Project Community. *Netty*. Netty Website, 2013. Retrieved October 2013, from `http://netty.io/`

[Crane and McCarthy 2008] Dave Crane and Phil McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, Berkely, CA, USA, 2008.

[Crane 2013] Charlie Crane. Building fast scalable game server in node.js. In *JSConf.Asia*, 2013.

[Crockford 2006] Douglas Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627, IETF, 7 2006.

[DeCandia *et al.* 2007] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[del Pilar Salas-Zrate *et al.* 2012] Mar del Pilar Salas-Zrate, Giner Alor-Hernndez, and Alejandro Rodrguez-Gonzlez. Developing lift-based web applications using best practices. *Procedia Technology*, 3(0):214 – 223, 2012. ¡ce:title¿The 2012 Iberoamerican Conference on Electronics Engineering and Computer Science¡/ce:title¿.

[Dierks and Rescorla 2008] Tim Dierks and Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*, 2008.

[Ding *et al.* 2003] Choon Hoong Ding, Sarana Nutanong, and Rajkumar Buyya. *Peer-to-Peer Networks for Content Sharing*. Technical report, Laboratory, University of Melbourne, Australia, 2003.

[Dukkipati *et al.* 2010] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing tcp's initial congestion window. *SIGCOMM Comput. Commun. Rev.*, 40(3):26–33, June 2010.

[Dye *et al.* 2009] Matthew W.G. Dye, C. Shawn Green, and Daphne Bavelier. Increasing speed of processing with action video games. *Current Directions in Psychological Science*, 18(6):321–326, 2009.

[Enix 2014] Square Enix. *Square Enix*. online, January 2014. Retrieved January 2014, from `http://www.square-enix.com/`

[Entertainment 2010] Blizzard Entertainment. *StarCraft II Players Banned*. Battle.net blog, September 2010. Retrieved December 2013, from `http://us.battle.net/sc2/en/blog/882508`

[Epic 2014] Epic. *http://epicgames.com/*. online, January 2014. Retrieved January 2014, from `http://epicgames.com/`

[Erb 2012] Benjamin Erb. *Concurrent Programming for Scalable Web Architectures*. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012.

[Fette and Melnikov 2011] Ian Fette and Alexey Melnikov. *The WebSocket protocol.* Technical report, Internet Engineering Task Force, 2011.

[Fidge 1988] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[Fiedler 2008] Glenn Fiedler. *Networking for Game Programmers*, 2008.

[Fielding *et al.* 1999] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, 1999.

[Fink 2012] Bryan Fink. Distributed computation on dynamo-style distributed storage: Riak pipe. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, pages 43–50, New York, NY, USA, 2012. ACM.

[Foundation 2010] Apache Software Foundation. *Apache HTTP Server Reference Manual - for Apache Version 2.2.17.* Network Theory Ltd., 2010.

[Foundation 2013] Apache Software Foundation. Apache jmeter. 2013. Retrieved October 2013, from `http://jmeter.apache.org/`

[Freier *et al.* 2011] Alan Freier, Paul Kocher, and Philip Karlton. *The Secure Sockets Layer (SSL) Protocol Version 3.0*, 2011.

[Gilbert and Lynch 2002] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[Gilbert 2012] David Gilbert. Diablo 3 server meltdown unable to meet demand. *International Business Times*, May 2012.

[Google 2012] Google. *SPDY: An experimental protocol for a faster web*, 2012. Retrieved October 2013, from `http://www.chromium.org/spdy/spdy-whitepaper`

[Griwodz and Halvorsen 2006] Carsten Griwodz and Pål Halvorsen. The fun of using tcp for an mmorpg. In *Proceedings of the 2006 International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '06, pages 1:1–1:7, New York, NY, USA, 2006. ACM.

[Gupta 2012] Munish K. Gupta. *Akka Essentials.* Packt Publishing, October 2012.

[Gustafsson 2005] Andreas Gustafsson. Threads without the pain. *Queue*, 3(9):34–41, November 2005.

[Habeeb 2010] Mocky Habeeb. *A Developer's Guide to Amazon SimpleDB.* Addison-Wesley Professional, 1st edition, 2010.

[Haller *et al.* 2013] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. *SIP-14 - Futures and Promises.* Scala Improvement Process Website, February 2013. Retrieved October 2013, from `http://docs.scala-lang.org/sips/pending/futures-promises.html`

[Hampel *et al.* 2006] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM.

[Harcsik *et al.* 2007] Szabolcs Harcsik, Andreas Petlund, Carsten Griwodz, and Pål Halvorsen. Latency evaluation of networking mechanisms for game traffic. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '07, pages 129–134, New York, NY, USA, 2007. ACM.

[Harrah 2013] Mark Harrah. *SBT*. github, 2013. Retrieved November 2013, from `http://www.scala-sbt.org/`

[Hewitt *et al.* 1973] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[Hewitt 2012] Carl Hewitt. *Actor Model of Computation: Scalable Robust Information Systems*. Technical Report v24, July 2012. cite arxiv:1008.1459Comment: improved syntax.

[Hollister 2014] Sean Hollister. *Sony announces PlayStation Now, its cloud gaming service for TVs, consoles, and phones*. TheVerge.com, 2014. Retrieved January 2014, from `http://www.theverge.com/2014/1/7/5284294/sony-announces-playstation-now-cloud-gaming`

[Honeywell 2000] Steve Honeywell. *Command and Conquer Red Alert 2*. Prima Communications, Inc., Rocklin, CA, USA, 2000.

[Howland 1999] Geoff Howland. *What is Lag?* Gamedev, September 1999. Retrieved October 2013, from `http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/what-is-lag-r712`

[Hu *et al.* 2006] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. VON: a scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22–31, July 2006.

[id 2014] id. *id Software*. online, January 2014. Retrieved January 2014, from `http://www.idsoftware.com`

[Iimura *et al.* 2004] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, pages 116–120, New York, NY, USA, 2004. ACM.

[Imtarnasan *et al.* 2012] Vibul Imtarnasan, Dave Bolton, and Socko Contributors. *Socko Web Server*. Socko Website, 2012. Retrieved October 2013, from `http://sockoweb.org/`

[Keller and Simon 2002] Joaquin Keller and Gwendal Simon. Toward a peer-to-peer shared virtual reality. In *In IEEE Workshop on Resource Sharing in Massively Distributed Systems*, pages 595–601, 2002.

[Kent 1998] Steven L. Kent. *Age of Empires*. Microsoft Press, Redmond, WA, USA, 1998.

[Knutsson *et al.* 2004] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 96–107. IEEE, 2004.

[Konishetty *et al.* 2012] Vamshi Krishna Konishetty, K. Arun Kumar, Kaladhar Voruganti, and G. V. Prabhakara Rao. Implementation and evaluation of scalable data structure over hbase. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 1010–1018, New York, NY, USA, 2012. ACM.

[Krasser 2013] Martin Krasser. *Eventsourced*. Github, 2013. Retrieved October 2013, from `https://github.com/eligosource/eventsourced`

[Krause 2008] Stephan Krause. A case for mutual notification: a survey of p2p protocols for massively multiplayer online games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '08, pages 28–33, New York, NY, USA, 2008. ACM.

[Kurose and Ross 2009] James F. Kurose and Keith Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley, Boston, MA, USA, 5th edition, 2009.

[Lakshman and Malik 2010] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[Lamport 1998] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Langley 2012] Adam Langley. *Transport Layer Security (TLS) Next Protocol Negotiation Extension*. Technical report, May 2012.

[Lauer and Needham 1979] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

[Lee 2012] Trustin Lee. *Netty 3.3.1 released - SPDY Protocol !* Netty News Archive, February 2012. Retrieved October 2013, from `http://netty.io/news/2012/02/04/3-3-1-spdy.html`

[Li and Zdancewic 2007] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, June 2007.

[LinkedIn 2012] LinkedIn. *Project Voldemort: A distributed database*. online, March 2012. Retrieved January 2013, from `http://project-voldemort.com/`

[Meenan 2012] Patrick Meenan. *WebPagetest - Website Performance and Optimization Test*, 2012. Retrieved July 2013, from `http://www.webpagetest.org/`

[Microsoft 2014a] Microsoft. *Age of Empires Online*. online, January 2014. Retrieved January 2014, from `http://ageofempiresonline.com`

[Microsoft 2014b] Microsoft. *Halo Official Site*. online, January 2014. Retrieved January 2014, from `https://www.halowaypoint.com`

[Midway 2014] Midway. *Midway Games Inc.* online, January 2014. Retrieved January 2014, from `http://www.midway.com`

[Mulholland and Hakala 2004] Andrew Mulholland and Teijo Hakala. *Programming Multiplayer Games*. Wordware Publishing Inc., Plano, TX, USA, 2004.

[Mulligan *et al.* 2003] Jessica Mulligan, Bridgette Patrovsky, and Raph Koster. *Developing Online Games: An Insider's Guide*. Pearson Education, 1 edition, 2003.

[Murphy 2011] Curtiss Murphy. Believable dead reckoning for networked games. In Eric Lengyel, editor, *Game Engine Gems 2*, pages 307–328. A K Peters, 2011.

[Netcraft 2013] Netcraft. *June 2013 Web Server Survey*. Netcraft Website, June 2013. Retrieved October 2013, from `http://news.netcraft.com/archives/2013/06/06/june-2013-web-server-survey-3.html`

[Neumann *et al.* 2007] Christoph Neumann, Nicolas Prigent, Matteo Varvello, and Kyoungwon Suh. Challenges in peer-to-peer gaming. *SIGCOMM Comput. Commun. Rev.*, 37(1):79–82, January 2007.

[Ng 1997] Yu-Shen Ng. *Designing Fast-Action for the Internet*, 1997.

[Nordwall 2013a] Patrik Nordwall. *50 million messages per second - on a single machine*. Let It Crash blog, 2013. Retrieved November 2013, from `http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine`

[Nordwall 2013b] Patrik Nordwall. *Durable mailbox as a default mailbox*. Akka Mailing List, 2013. Retrieved October 2013, from `https://groups.google.com/d/topic/akka-user/X_CI9fafRmU/discussion`

[Olson *et al.* 1999] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[Ousterhout 1996] John Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Winter Technical Conference*, January 1996.

[Pachev 2007] Sasha Pachev. *Understanding MySQL Internals*. O'Reilly Media, Inc., 2007.

[Plugge *et al.* 2010] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[Rabl *et al.* 2012] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, August 2012.

[Richmond and Williams 2011] Shane Richmond and Christopher Williams. Millions of internet users hit by massive sony playstation data theft. *The Telegraph*, 2011.

[Ridgway 2011] Jamie Ridgway. Actor-based programming. In *Code PaLOUsa*, 2011.

[Ritchie 2013] Brian Ritchie. *RavenDB high performance*. Packt Publ., Birmingham, 2013.

[Rivest 1992] Ron Rivest. *The MD5 Message-Digest Algorithm*, 1992.

[Rob *et al.* 2007] Peter Rob, Carlos Coronel, and Keeley Crockett. *Database Systems: Design, Implementation and Management*. Course Technology Press, Boston, MA, United States, international edition, 2007.

[Rowstron and Druschel 2001] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[Sadalage and Fowler 2013] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled : a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, Upper Saddle River, NJ, 2013.

[SBT 2013] SBT. *sbt-assembly*. github, 2013. Retrieved November 2013, from `https://github.com/sbt/sbt-assembly`

[Schiele *et al.* 2007] Gregor Schiele, Richard Sueselbeck, Arno Wacker, Torben Weis, Joerg Haehner, and Christian Becker. Requirements of peer-to-peer-based massively multiplayer online gaming. In *Proceedings of the Seventh International Workshop on Global and Peer-to-Peer Computing, organized at the IEEE/ACM International Symposium on Cluster Computing and the Grid 2007, CCGRID 2007*, Rio de Janeiro, Brazil, May 2007.

[Schroeder 2013] Kevin Schroeder. *Why is FastCGI /w Nginx so much faster than Apache /w mod_php?* ESchrade blog, 2013. Retrieved October 2013, from `http://www.eschrade.com/page/why-is-fastcgi-w-nginx-so-much-faster-than-apache-w-mod_php/`

[Schulzrinne *et al.* 2003] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*, 2003.

[ServerStack 2012] ServerStack. *5 Ways To Speed Up Your Website With Nginx Web Server*. online, 2012. Retrieved October 2013, from https://www.serverstack.com/blog/2012/02/07/5-ways-to-speed-up-your-website-with-nginx-web-server/

[Shea *et al.* 2013] Ryan Shea, Jiangchuan Liu, Edith C. H. Ngai, and Yong Cui. Cloud gaming: Architecture and performance. *IEEE Network*, 27(4), 2013.

[Silberschatz *et al.* 2008] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[Sissel 2010] Jordan Sissel. *SSL handshake latency and HTTPS optimizations*. semicomplete website, June 2010. Retrieved October 2013, from http://www.semicomplete.com/blog/geekery/ssl-latency.html

[Stewart *et al.* 2006] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen. *Stream Control Transmission Protocol (SCTP) Specification Errata and Issues*. RFC 4460 (Informational), April 2006.

[Stoica *et al.* 2001] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[Sumbaly *et al.* 2012] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[Suszek 2013] Mike Suszek. *Alliance wins The International 2013 Dota 2 tournament, earns over $1.4 million*. Joystiq Website, August 2013. Retrieved December 2013, from http://www.joystiq.com/2013/08/12/alliance-wins-the-international-2013-dota-2-tournament-earns-ov/

[Thomas *et al.* 2012] Bryce Thomas, Raja Jurdak, and Ian Atkinson. Spdying up the web. *Commun. ACM*, 55(12):64–73, December 2012.

[Tilkov and Vinoski 2010] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, November 2010.

[Typesafe 2013] Typesafe. *Akka Scala Documentation*. Typesafe Inc, 2.2.1 edition, August 2013.

[Valve 2012] Valve. *Source Multiplayer Networking*. Valve Developer Community, 2012. Retrieved 1 July 2012, from https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

[van der Lans 2010] Rick F. van der Lans. *InfiniteGraph: Extending Business, Social and Government Intelligence with Graph Analytics*. Technical report, R20/Consultancy, September 2010.

[Venners 2011] Bill Venners. *Writing Concurrent, Scalable, Fault-Tolerant Systems with Akka 1.0*. Artima Developer Website, 2011. Retrieved June 2013, from http://www.artima.com/scalazine/articles/akka_jonas_boner.html

[Virkki 2013] Jyri J. Virkki. *Using node.js for serving static files*. online, 2013. Retrieved December 2013, from http://www.virkki.com/jyri/articles/index.php/using-node-js-for-serving-static-files/

[Vogels 2009] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[von Behren *et al.* 2003] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[W3Techs 2013] W3Techs. *Usage of web servers for websites*. W3Techs Website, 2013. Retrieved October 2013, from `http://w3techs.com/technologies/overview/web_server/all`

[Wang *et al.* 2012] Vanessa Wang, Frank Salim, and Peter Moskovits. *The Definitive Guide to HTML5 WebSocket*. Apress, December 2012.

[Webber 2012] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 217–218, New York, NY, USA, 2012. ACM.

[White 2007] Tom White. *Consistent Hashing*. Java.net blogs, November 2007. Retrieved October 2013, from `https://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.htmll`

[Xu *et al.* 2009] Ke Xu, Meina Song, Xiaoqi Zhang, and Junde Song. A cloud computing platform based on p2p. In *IT in Medicine Education, 2009. ITIME '09. IEEE International Symposium on*, volume 1, pages 427–432, Aug 2009.

[Yahyavi and Kemme 2013] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):9:1–9:51, July 2013.

[Yamamoto *et al.* 2005] Shinya Yamamoto, Yoshihiro Murata, Keiichi Yasumoto, and Minoru Ito. A distributed event delivery method with load balancing for mmorpg. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, NetGames '05, pages 1–8, New York, NY, USA, 2005. ACM.

[Yu *et al.* 2012] Su-Yang Yu, Nils Hammerla, Jeff Yan, and Peter Andras. Aimbot detection in online fps games using a heuristic method based on distribution comparison matrix. In *Proceedings of the 19th International Conference on Neural Information Processing - Volume Part V*, ICONIP'12, pages 654–661, Berlin, Heidelberg, 2012. Springer-Verlag.

[Ziebart 2011] Alex Ziebart. World of warcraft dips to a mere 11.4 million subscribers. *WoW Insider*, 2011, May 2011.