Peer-to-peer Game State Replication

A practical application of the Same platform

Kjetil Mehl

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Multiplayer mobile games using the local network is a tempting platform, however, several technical hurdles remain in order to create this technology. The *Same* framework by Kjetil Ørbekk applies techniques from distributed systems in the pursuit of creating a platform for sharing objects among Android devices on a local network

Throughout this thesis, a multiplayer game design was devised and implemented using Same's transmission protocol, and the master selection mechanism for improved connection stability. The multiplayer game provides a working model for analyzing Same's features and limitations for practical application.

The game's network architecture uses a flexible client-server methodology. The acting master device maintains the server role. If the master device fails, a new master device is selected, which resumes the simulation from the last known game state. Snapshots and delta states are managed in a replication model in order to reduce lag and improve reliability.

This thesis focuses on ways to adapt the Same framework to gaming platforms, including its strengths and limitations. The major strength in Same is that its updates are consistently distributed to all connected clients. This could also pose as one of its major weaknesses for games that do not have to propagate state this strictly. Several elements of the Same platform are further analyzed, whereby additional research and development could transform Same into a strong distributed local networking platform.

Samandrag

Mobilspel med støtte for fleire deltakarar samstundes over lokalnettet er ein attraktiv plattform, men det gjenstår framleis mange problem i å utvikle slik teknologi. Rammeverket *Same* utvikla av Kjetil Ørbekk brukar teknikkar frå distribuerte system for å lage ein plattform med føremål å dele objekt mellom Android-einingar på eit lokalt nettverk.

Gjennom denne avhandlinga har det blitt utforma og utvikla ein prototype som brukar protokollen i Same for å formidle kontakt mellom klientar samt mekanismen for å automatisk oppretthalde kontakta mellom klientane. Spelet tilrettelegg for ein modell som gjer det mogeleg å analysere Same sine eigenskapar og avgrensingar.

Nettverkarkitekturen i spelet er bygd på ein fleksibel klient-tenar metodikk. Android-eininga som opptrer som fungerande master vil og få rolla som tenar. Om denne masteren feilar vil ei ny eining bli valt som master og simuleringa vil halde fram frå den sist kjende tilstanden. Augneblinksbilete av tilstanden og deltatilstander blir handtert i ein replikeringsmodell for å minske etterslep og for å auke pålitelegheita.

Denne avhandlinga fokuserer på måtar å tilpasse Same-rammeverket til bruk i spel både med tanke på rammeverket sine styrkar og avgrensingar. Den største styrken i Same er at oppdateringane konsistent vert distribuert til alle klientane. Dette kan vise seg å verte ein veikskap ved systemet for spel som naudsynt ikkje har behov for denne konsistensen. Ulike aspekt ved programvareplattforma er analysert og vidare utvikling på rammeverket kan forbetre Same til ein endå sterkare kandidat for distribuert tilstand over lokale nettverk.

Acknowledgements

I would like to thank the following people for their help and input during the project:

- **Svein Erik Bratsberg** as my supervisor. Svein Erik has given valuable and insightful feedback throughout the whole project. He has enthusiastically answered my questions and helped me with implementation problems and design choices.
- **Jonas Eikli** who provided me with an additional test device. Jonas also helped me record and create the demonstration video.

Contents

List of Figures

List of Tables

Glossary

Android Android. 56

API Application Programming Interface. 11, 13

CPU Central Processing Unit. 51, 56

DDMS Dalvik Debug Monitor Server. 56, 57

DHCP Dynamic Host Configuration Protocol. 8

DNS Domain Name System. 8

FPS First Person Shooter. 6, 25, 50, 52, 56, 71, 72

GC Garbage Collector. 19, 57

GPU Graphic Processing Unit. 36, 56

GWT Google Web Toolkit. 35

HTTP Hypertext Transfer Protocol. 13

IP Internet Protocol. 7, 29, 34, 40, 57

IPX Internetwork Packet Exchange. 7

JSON Javascript Object Notation. 9, 12, 41–43, 63

JXTA Juxtapose. 12

LAN Local Area Network. 6, 7, 12

MVC Model View Controller. 18, 19, 35

NAT Network Address Traversal. 12, 13

NTNU Norwegian University of Science and Technology. 8

OpenGL Open Graphics Library. 35, 39

RDV Rendezvous. 13

RTS Real Time Strategy. 6, 7, 24

SDK Software Development Kit. 56, 59

SPX Sequenced Packet Exchange. 7

TCP Transmission Control Protocol. 7–10, 12, 13

UDP User Datagram Protocol. 7, 8, 12, 13, 61, 62

UML Unified Modeling Language. 21

Wi-Fi Wireless. 1, 7

XML Extensible Markup Language. 12

Chapter 1

Introduction

Networked games have played an important part throughout gaming history. Before the widespread of always-on Internet it was not unusual for people to host Local Area Network parties and computer gatherings. As more people got connected to the Internet this tendency shifted over to online gaming enabling massive online multiplayer games enabled by powerful server parks around the globe.

When smartphones were introduced to the market casual gaming developed as a whole new market. These games are often small scale productions meant to entertain the player over a short time span. Smartphones have risen to be powerful devices capable of displaying advanced graphics and perform heavy computations. By combining this technology with an always-on connection to the Internet a great base for real-time multiplayer games on smartphones is created.

Same is a framework developed especially for intercommunication between Android devices[10]. These devices are usually connected to Internet through unreliable channels such as cellular[1] or Wireless (Wi-Fi)[2] networks. Packet loss, connection loss and unreachable hosts are occurring more frequent in these networks compared to traditional cabled network. If the host in a game loses connection all connected clients will suffer and the current game state will be lost. This is the main problem that Same tries to solve. Its solution is based on letting all connected clients select a new 'master' if the prior fails. Once a consensus has been established, the new master takes on the responsibility of distributing state to all connected clients.

[1]Mobile network - `http://en.wikipedia.org/wiki/Mobile_network`
[2]Wi-Fi - `http://en.wikipedia.org/wiki/Wi-Fi`

1.1 Problem Statement

When Same was developed the intended application was real-time multiplayer games running on mobile devices. Same is capable of connecting clients and facilitating communication between those clients.

However, many challenges are faced when creating a real-time multiplayer game. What kind of network architecture should be applied? What kind of data should be shared between connected clients? What could be considered a reliable source of this data, and how should this data be handled by a receiving client?

Several of these challenges are dependent on the performance of the underlying network implementation. In a typical real-time game updates must be sent and received at a high and steady rate in order to keep the state between all connected clients synchronized. The amount of data to be sent, how many clients should receive the update and the latency to these clients are all important factors that could have a major impact on the overall performance of the system.

The main focus of this thesis is to see how well Same handles state propagation of a typical real-time game. In networked multiplayer games it is crucial to minimize the amount of state that needs to be propagated. Passing data through the network is an expensive operation, and it is desired to minimize the amount of redundant data. There will also be a focus on maintaining a smooth gameplay experience for connected clients even if intermediate connection losses or delayed game updates occur.

1.2 Project Goal

The goal of this thesis is to adapt the capabilities and the functionality in Same to see how well it performs as a platform for real-time multiplayer games. To achieve this goal a multiplayer game with real-time requirements will be devised and implemented. Same will be used as the prototype state replicator, and the game should exploit the fault detection and fault recovery provided in this framework. The prototype should be modeled after typical real-time multiplayer games in order to get a realistic test case and realistic results.

A part of this goal is to propose solutions to problems encountered throughout the implementation phase both related to game network implementation generally, and the adaption of Same specifically for this game prototype.

1.3 Thesis Outline

Chapter 2 of this report aims to provide some background literature on real-time multiplayer games and network gaming. This chapter will also give a short introduction to Same. Chapter 3 describes several existing solutions, based on different network models. Chapter 4 aims to describe the concept of the game to be implemented and its architectural design. Chapter 5 covers the implementation phase which includes libraries that was used and various challenges encountered. Chapter 6 aims to evaluate the proposed solutions. Conclusions and further work are presented in Chapter 7.

Chapter 2

Background

This chapter aims to describe the background behind the project. The main goal is to create a prototype of a multiplayer game on top of a specialized network model. This chapter will give some background on what a multiplayer game is and how nodes in a typical networked game are connected. It will also give a quick introduction to Same, the network model that is to be adapted.

2.1 Real-time Multiplayer Games

Multiplayer games are based on two or more participating players. This is opposed to single-player games whereas a player is playing against the artificial intelligence of the computer.

People are often more involved in games where the opponent is controlled by another human being. A study on competitiveness in games[12] showed that players battling other players fostered a social competition. The research argued that competitive elements can be incorporated by such games because of their interactivity, which allows for active engagement of the user in the playing process, and for feedback on user's actions. Further they stated that the user's feeling to play against an opponent likely evokes a social-competitive situation, that is especially capable to engage and involve the users.

However there are several drawbacks by designing and implementing a multiplayer game. The game itself will in most cases be more complex than a single-player game. Players must be able to interact with the game individually. The gameplay must be balanced, and every participating player must feel that they have a chance to win.

The grand slam of multiplayer games are networked games. These are often games running over the Internet, an unreliable, unstable and limited communication channel. The game state must be replicated over this channel, players must synchronize their simulations and connections losses must be handled. Greater problems arises when the server that simulates the game world crashes. How to retrieve the game state? These are just a handful of the many problems that a networked game must account for.

Two extremities are often mentioned when speaking about design of networked games, namely *real time* and *turn based* design. The scope of this thesis is to focus on real time games, and its informal definition from Wikipedia[14] is given below:

> *In real-time games, game time progresses continuously according to the game clock. Players perform actions simultaneously as opposed to in sequential units or turns. Players must perform actions with the consideration that their opponents are actively working against them in real time, and may act at any moment. This introduces time management considerations and additional challenges (such as physical coordination in the case of video games).*

One method to implement a networked game is to progress the game clock at each client once the client receives a packet signed with the next tick from the server, or the other clients. This is referred to as Lockstep progression[13]. Some games have implemented this system (see Section 4.5.1), but it's far to error prone for wireless communication channels which most Android devices use. In addition to this the game time would only progress as fast as the client with the highest latency since its response time would dictate when the next game state could be achieved.

Many real-time networked games turn to a client-server approach. Instead of progressing the game clock in parallel, each client progresses its internal game clock and updates its state based on an authoritative source. This source might be another client or a dedicated server. Grand examples of real-time multiplayer games that have adapted this model are:

- Quake (First Person Shooter (FPS))
- Counter Strike (FPS, Half Life modification)
- Starcraft 2 (Real Time Strategy (RTS))

Quake and Counter Strike are typical real time multiplayer games utilizing the client-server model. These games are implemented such that every player that creates a game, starts a new server and connects to itself as a client. This model enabled people to create games on the Local Area Network (LAN), without being dependent on a Internet connection. It also enabled people to host their own servers on the Internet, creating a self driven game service.

Starcraft 2 also uses the client-server model, except that all players are restricted to being clients only. The servers are controlled entirely by the game company. Drawbacks with this model is that two players are unable to finish their game if the centralized servers experience problems. People are also forced to play through the Internet, even when they are located on the same LAN. Advantages with this model is that Blizzard are able to control every aspect of the simulation, greatly reducing probability of cheaters and mischievous players. The company needs a large, stable and secure infrastructure to support this kind of model.

2.2 Networked Games

There exists several methods of achieving a multiplayer enabled game. Methods such as "split screen" and "hot swapping" was popular some years back, due to the poor availability of "always connected" Internet. In the later years networked games have gained popularity, due to the larger user base these games and the Internet provide. This thesis will concentrate on networked games, so it is natural to expand on what defines a networked game. The 5-layer TCP/IP model in Table 2.1 will be used to explain connectivity at different levels.

A networked game shares state between its players over a communication link. This link can be everything between a network switch or router, a wireless access point/hotspot or a mobile base station. Regular desktop computers are usually connected through a cable to a switch or a router. Mobile devices on the other hand are most often exclusively equipped with a Wi-Fi antenna and a mobile radio. Laptops often have all three options. Each of these peripherals implement specific network protocols (layer 2 in Table 2.1).

Application	*(layer 5)*
Transport	*(layer 4)*
Internet	*(layer 3)*
Network Interface	*(layer 2)*
Link	*(layer 1)*

Table 2.1: The 5-layer TCP/IP reference model

The present de facto communication protocol over these unreliable channels are the Internet Protocol (IP) (layer 3) and Transmission Control Protocol (TCP) (layer 4). During the 1980s through to the mid-1990s two other protocols were popular on LAN. The Internetwork Packet Exchange (IPX) (layer 3) and Sequenced Packet Exchange (SPX) (layer 4). The popular RTS game Warcraft 2 for instance only supported this technology. IPX/SPX was proven to perform faster than TCP/IP in smaller networks, but due to the latter performing far superior on Wide Area Networks[1], IPX/SPX gradually got phased out.

Nowadays most networked games rely on either TCP, User Datagram Protocol (UDP) or both protocols. These are described in the following sub sections.

[1]Wide Area Networks - `http://en.wikipedia.org/wiki/Wide_area_networks`

2.2.1 TCP

TCP[2] provides a reliable, ordered, error-checked delivery of a stream of octets between clients connected in a network. TCP is used as the main protocol for data transport for the World Wide Web, e-mail and file transfers.

Using TCP as the main transport protocol for games, simplifies the implementation for the developers, but may create issues in regards to performance later on. There is considerably overhead of transmitting a TCP packet in contrary to a UDP packet. A chat system within a game would be a good candidate for a TCP based implementation. The chat messages would most likely be small in terms of size, and they would not be sent that frequent. Players would also expect that a message they sent actually got sent. This is something TCP helps to ensure.

2.2.2 UDP

UDP[3], is a "hit or miss" protocol, meaning that it does not care whether or not packets did arrive. The main use case for UDP over TCP is performance and speed. It is a simple and stateless protocol very suitable for broadcasting information. It is used as the transport protocol for Domain Name System (DNS), Dynamic Host Configuration Protocol (DHCP) and often real time video and audio streaming.

Fast paced multiplayer games needs to exchange data at an rapid rate (maybe as often as a 100 updates per second). In these cases using TCP will not be sufficient. The desired update rate cannot be achieved due to latency in the network, and the overhead by using that protocol can be too large. The drawback by using UDP is that packets need to be verified at the end points (the clients), and this logic needs to be implemented manually by the developer.

2.3 Same

Same is a framework that was developed as a master thesis by a MsC student at the Norwegian University of Science and Technology (NTNU) during the spring of 2012. The main goal in this thesis was to propose a general solution to network failures in distributed systems for mobile phones. More specifically real-time multiplayer games.

Same has several valuable properties which often are desired in networked multiplayer games. Communication between connected clients are facilitated through a shared state. This means that every client can broadcast a change to the other clients. In addition to this the system knows what clients are connected at any time. This is usable for typical games where a player state needs to be maintained.

[2]TCP - `http://en.wikipedia.org/wiki/Transmission_Control_Protocol`
[3]UDP - `http://en.wikipedia.org/wiki/User_Datagram_Protocol`

The author of Same performed several benchmarks to evaluate the platform. The results discovered in this evaluation can be viewed in detail in his thesis but the main points are summarized below.

- The framework has reasonably low latency. The author predicts that it may be good enough for real time applications, as long as concurrent transferred objects is kept to a minimum.
- Same appears to scale well. With more clients though more updates are expected, which could lead to poor performance.
- The master selection routine is quickly able to select a new master once the prior fails.

By looking at the conclusion in the master thesis it is fair to say that Same is a good candidate as a framework for real-time network games.

2.4 The Network Model in Same

The model that Same proposes is different from both the traditional client-server model and the peer-to-peer model. It is more fair to say that Same is a hybrid between those two models. If two or more clients are joined in a network, they share state through a concept named *Variables*. When a client updates a Variable this update is propagated through Same, and all other connected clients gets updated. The propagation happens through the *Master* in this network.

A *master* is chosen by all participating clients. This selection routine is carried out by an implementation of the Paxos protocol[7]. The chosen master is responsible for facilitating contact between clients (notifying all clients when an update to a Variable occurs). This is similar to the client-server model where a central server is the communication channel for the clients. If the master becomes unavailable or disconnects from the network, the remaining participants selects a new master, and communication continues from the last known state. This model is comparable to the peer-to-peer model where all peers are equally responsible for facilitating communication. Clients must attach listeners for its variables in order to get notified when they are updated. A high level description of the Same model is given in Figure 2.1.

The underlying implementation of how state is propagated in Same is best explained by the use of Table 2.1. When a Same Variable gets set (on the Application level) to a Java object, this object is serialized into a Javascript Object Notation (JSON) string. The serializing is done by the JSON processor Jackson. Next the state at a client (including the JSON String) is serialized by using Protocol Buffers into a binary format. Then Same writes the encoded data to a Java Socket. When Same gets initialized on an end point a network socket over TCP (on the Transport level) gets initiated between the master and the client. When looking at the

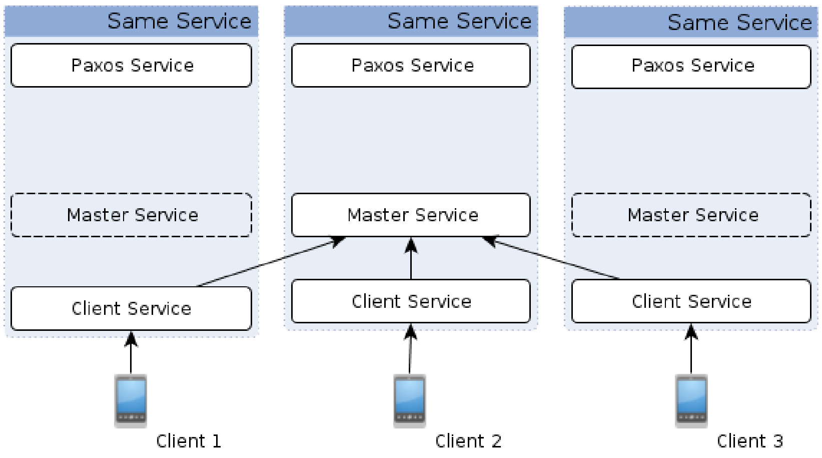


Figure 2.1: High level view of a Same network. A master gets elected by the participating clients, and this master facilitates contact between all clients. All clients individually run a Paxos service which triggers when the connection to the current master is lost.

premise for creating Same, using TCP makes a lot of sense. The main applications rely on the data being successfully transmitted which TCP handles (see Section 2.2.1). Same ensures that the correct state is propagated to all the connected clients.

Chapter 3

Related Work

In order to propose a successful game architecture existing solutions need to be investigated and researched. This research focuses on open source network game frameworks mainly intended for Android or Java in general. The main reason to look into open source[1] frameworks is that the code base is already freely available on the Internet.

3.1 Client-Server Model

Client-server is a distributed network paradigm that defines a set of clients connected to one centralized server. If this server fails or becomes unreachable, the clients will lose their connection as well. This model is illustrated in Figure 3.1.

3.1.1 KryoNet

KryoNet[2] is a lightweight client-server library built on top of Java Sockets. This library provides a simple Application Programming Interface (API) for setting up a server in one node, and for connecting to this server from other nodes. In the default implementation these nodes communicate by passing Java Objects to each other.

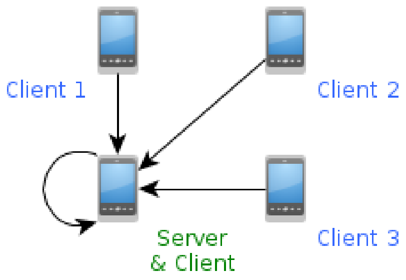


Figure 3.1: The client-server model.

[1] Open Source - http://en.wikipedia.org/wiki/Open_source
[2] KryoNet - http://code.google.com/p/kryonet/

By default KryoNet uses Kryo for serialization. Kryo uses a binary format and is very efficient, highly configurable, and does automatic serialization for most object graphs. It is also possible to plug in custom serialization (for example JSON). A prerequisite for using Kryo is that all classes which at one point should be passed through KryoNet have to be registered internally in the Server instance and the Client instance.

As with regular Java Sockets KryoNet provides communication over TCP, UDP or both protocols concurrently. A small set of utilities is also included in KryoNet such as server discovery on the LAN (broadcasts an UDP packet) and network latency testing by sending ping packets to connected clients.

3.2 Peer to Peer Model

Peer-to-peer is a distributed computing paradigm that defines a network architecture based on cooperating participants (peers). This model is illustrated in Figure 3.2.

3.2.1 Peerdroid

Peerdroid[3] is a port of the Juxtapose (JXTA) protocol to the Android platform. This protocol is an open source peer-to-peer protocol specification introduced by Sun MicroSystems. This protocol is defined in terms of a set of Extensible Markup Language (XML) messages.

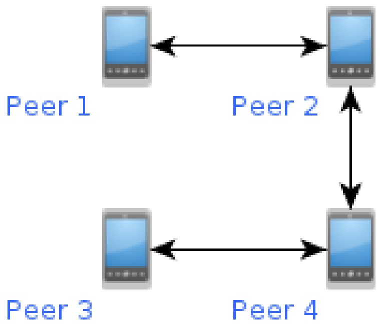


Figure 3.2: The peer to peer model.

There are two main categories of peers in this network. The edge peers and the super-peers. The edge peers are defined to be the peers which have transient, low bandwidth network connectivity. The reside on the border of the Internet, behind Network Address Traversal (NAT) or firewalls.

The super-peers are peers facilitating communication and discovery in this network. A *rendezvous peer* is in charge of coordinating the peers, and provides the necessary message propagation. A *relay*

[3]Peerdroid - `http://code.google.com/p/peerdroid/`

peer allows peers which are behind firewalls or NAT to take part in this network. This is done by using a protocol which can traverse the firewall, like Hypertext Transfer Protocol (HTTP).

This library creates a network based on an index of peers, called the Rendezvous (RDV) list. Peers connect through pipes (an abstraction similar to Unix pipes) which are built on top of Java sockets. These pipes can be direct communication two peers, or routed trough a rendezvous peer.

3.3 Hybrid Solutions

This section will describe a solution that is partly server-client based and partly peer-to-peer based. This is the type of solution which would be closest to Same feature set wise.

3.3.1 Google Play Game Service

At the annual Google IO conference held in 2013, Google announced a game service for Android and iOS (previously named iPhone OS). This platform includes a multiplayer API specifically targeted for Android. The goal with this platform is to make it easier for developers to create real-time multiplayer games.

This is a new type of service offered by Google and not much is known about its internal design and architecture. Some time was invested in trying out this service and it turns out that it is fairly similar to Same. The service is comprised of a high level API for finding matches, creating quick games and to invite friends to these games. One or several peers can be joined in a room and all peers are able to pass messages to other peers, or broadcast messages to the whole room. A room is similar to the network in Same and messages sent by these players is similar to how Same distributes its state. All communication between peers is authenticated by Google and conveyed through their servers. This is similar to the traditional client-server architecture. Peers can communicate by sending each other *UnreliableRealtimeMessages* or *ReliableRealtimeMessages*. The first would be an abstraction on top of UDP and the latter an abstraction on top of TCP. It is also possible to open direct sockets between the clients, which can be read and written to as one would do with the traditional Java API.

The game implementation and network logic is implemented on top of this API and how this implementation is realized is entirely up to the developer. The peer-to-peer network model could be exploited, or the developer could device a method to select one of the peers as an authoritative source of data much like the client-server-architecture. It is important to emphasize that this is a service provided by a third party. If the service gets shut down so will all games using this service as their multiplayer platform.

Chapter 4

Design

The aim of this chapter is to define and describe the design for this prototype.

4.1 Concept

This thesis is revolved around the capabilities of the Same framework, hence not a lot of time will be spent working on the game concept itself. There are some qualities by the game design required in order to properly test the framework. These are:

1. The game must be playable concurrently by several players (multiplayer)
2. The game must be real time, meaning that the game simulation progresses linearly with time.
3. There must be a shared state between all the players. This state could be the level representation, with dynamic entities such as players.
4. The prototype should not be too complex in terms of implementation
5. When a player interacts with the game, this interaction should be reflected as soon as possible

A short summary of how the concept for this prototype was devised is described in the following sub sections.

4.1.1 Devising a Game Concept

These requirements were used to brain storm around a simple game concept. I started out with a plan to implement a distributed version of the game Snake[1] (which later on became popular on mobile devices).

After experimenting for some time with this concept I realized it would not adhere to some of the requirements in the way I wanted. The game would not be as real time without breaking the classic Snake gameplay. Brainstorming with fellow students led to another concept based on user controlled marbles. These marbles should be able to move around in some sort of static level. One goal in this game concept would be to push the other marbles of a platform, or to be the first marble to reach a specific checkpoint. The game should be created top-down, meaning that the players see their marbles from a bird perspective. The accelerometer in the Android devices will be used as input. When a player tilts its device in one direction, the marble should roll in that direction.

This concept adheres to the specified requirements. The player states have to be synchronized real-time. All clients must at all times be aware of the other players. The level itself will mostly be static and comprised of walls and other obstacles. Physics will be an important aspect in this game since the players will use their momentum to push other entities. The game should allow for clients to connect and disconnect at arbitrary times. In the most basic use case two players are battling each other on a flat surface. In this case a simulation needs to calculate the positions, velocities and momentum for these marbles. This information must be sent to both these clients, which will get reflected on their devices. A hand drawn sketch of the game design is illustrated in Figure 4.1.

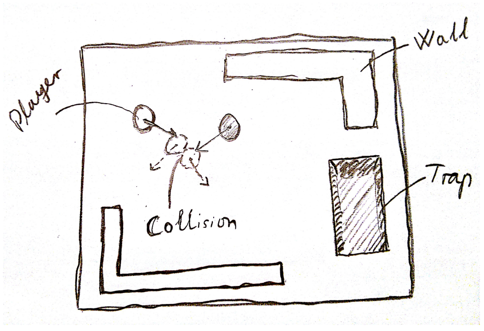


Figure 4.1: An initial game sketch. Two players (the marbles) are colliding with each other. Two walls and a trap are also present on the level. The game is seen from a top-down perspective.

4.1.2 Interaction Matrix

A game matrix is a common method to describe relationships between entities in a game design. In this design a game entity is an in-game object with one or several different tasks. The marble would be one such entity and has a task of moving around on the game level. Interactions between game entities are on a

[1]Snake - `http://en.wikipedia.org/wiki/Snake_%28video_game%29`

Entity					
Marble	✔				
Wall	✔				
Pit	✔				
Trap	✔				
	Marble	Wall	Pit	Trap	**Entity**

Table 4.1: Interaction matrix for the game design.

conceptual level and not restricted to physical collisions. Table 4.1 describes these relationships.

As this matrix displays the player controlled marble should interact with several other game entities. This table also describes something about the collisions and how these should be handled. A marble colliding with a wall should be bounced back with an equal but opposing force. A marble interacting with a pit on the other hand, should result in a player losing his marble. Note that the other defined entities should not interact with each other. This matrix will be used as a guideline throughout the implementation.

To keep the implementation simple the collisions and movement should happen in a two dimensional space. This makes sense since the game is designed to be top-down. Only having two degrees of freedom will also keep down the amount of data that have to be synchronized.

4.2 Architectural Requirements

This section will list the major architectural requirements for this system. The Rational Unified Process by IBM gives the following definition for any requirement[4]:

> *A requirement describes a condition or capability to which a system must conform; either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document.*

An architectural requirement is defined as any requirement that is architectural significant. By using the concept devised in Section 4.1 a set of architectural requirements are specified. Some of these requirements are directly targeted to test Same, while others are specified to ensure a fluent game experience. The requirements are listed with an identifier, a name and a short description.

AR #1 Localized simulation
In this system disconnects and errors can and will occur. The game experience can greatly be affected if the game stops or gets delayed because of intermediate errors. This should be solved by having every device in the network run an independent simulation.

AR #2 Authoritative master
As the aforementioned requirement states all clients run a local simulation. These simulations should be updated accordingly to the simulation running at the master. This ensures that the clients will be updated with the same data set, and that they will synchronized.

AR #3 Transient fail over
If the master gets disconnected or crashes, one of the clients should get selected as the new master and run the simulation. This should happen automatically.

AR #4 Minimal network traffic
Mobile phones are often connected to the Internet through a mobile data connection (such as 3G). These subscriptions are expensive compared to a regular connection, and the pay model is tied to data usage. It is therefore desired that the system minimizes network traffic.

AR #5 Gameplay based on physics
The gameplay is revolved on mechanical physics, forces and responses to collisions. This needs to be replicated on all devices in order to give a proper representation of the game world.

These requirements will be used as the basis when designing the architecture as seen in the following section.

4.3 Architecture

This section will describe the proposed architecture for this project. To create a clean and well designed architecture it is natural to expand on already existing design patterns and solutions. Software architecture patterns is one such way of defining general properties for the system in whole. By following a pattern all involved developers should immediately understand what qualities are expected of a module or component in this architecture.

The architectural pattern Model View Controller (MVC) first described by Trygve Reenskaug makes a good foundation for a game architecture. Its main purpose is to separate and provide a clean interface between user interface and data models[9]. As the name entails this is a three part architecture comprised by:

Model: stores object data and notify views when this data changes.

View: represents data stored in the models. The view is notified with changes from the models, and updates accordingly.

Controller: sends commands to its associated view to make the view change the view's representation of the model.

In a typical game the player object is stored as a model, represented by some mean of rendering (the view), and interacted with through the controller. MVC will be the underlying architectural pattern for this prototype. In the implementation this pattern will be adhered to by creating a GameScreen class as the view. This class will render the game entities (which are models). The controller will be the binding between the models and thew view. In this case the controller will pass player input to the models. The view will in response to this draw the updated models.

Another way to create a contract between the developer and the architecture is to use design patterns. A design pattern is defined by Erich Gamma et al.[6] as a "general reusable solution to a commonly occurring problem". The use of design patterns simplifies code complexity and enhances mutual understanding of the architecture between developers. For this kind of application there are several design patterns that can be applied. Examples of relevant design patterns are described below:

Singleton: Declaring a class as a singleton means that the class can't be instantiated multiple times during the program execution. Functionality can be protected behind the singleton in order to ensure that all calls to the its methods are handled by the one and only instance of that class. Reading data from files is an expensive operation and it is better to cache the data within the singleton, to prevent any unnecessary read operations.

Object pool: Object pools are a mechanism to limit new instances of objects. Instead of letting the program allocate new objects, it pulls objects from a pool of pre-allocated objects. Once the object has finished its task (gets de-referenced), it goes back into the pool. This design pattern is especially useful in Java, where all memory allocations are handled dynamically by the Garbage Collector (GC). On less powerful devices it could be noticeable when the GC runs.

Builder: This pattern separates the construction of a complex object from its representation allowing the same construction process to create various representations. This pattern is already being used in the code base of Same.

4.3.1 High Level Architecture

This section will describe the high level architecture for this system. An illustration of the initial design is given in Figure 4.2. Only one connected client is included for readability reasons.

To consider Figure 4.2 in the context of the MVC architecture the *headless* simulation will contain the correct state of all the models. These states are distributed to all connected clients which renders these models in their local view. Each client manipulates the headless simulation through an instance of the GameClient which is the controller.

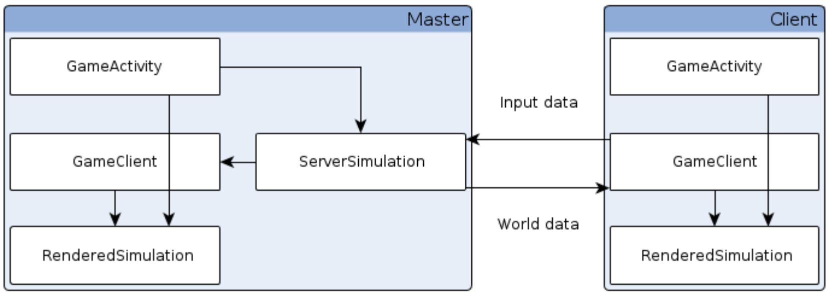


Figure 4.2: The conceptual view of one device running as master and another device running as client. The internal boxes describe class instances and the arrows between them describes the data flow.

This design is rather common when creating multiplayer games based on the client-server architecture (clients display the view, the server maintains and distributes the game state). The goal is to let one of the nodes in the network (the headless simulation) do actual computations and game state updates while the clients (the rendered simulation) only shadow this state and presents it for the players.

The headless simulation simulates the logic in this system. This includes collision detection, game entity updates, checking game rules and handling user input from the clients. This simulation has no concept of *how* this logic should be represented.

The rendered simulation renders objects for the players. These objects get updated from the headless simulation. The rendered simulation does not handle input from the players, but indirectly reflects the input when entities are updated in the headless simulation. This conforms with Architectural Requirement #2 (see Section 4.2).

4.4 Game Architecture

This section will describe the designed architecture for the game itself. This would be the architecture running at each client, independently on how those clients communicate. A diagram displaying the game architecture is best shown in the context of the rendered simulation in Figure 4.3. This figure is a detailed sub set of the diagram displayed in Figure 4.2.

Noteworthy in this figure are the methods in the rendered simulation. The *GameClient* object polls player input from the simulation and updates its internal variable. The game state from the master is received in the *GameClient* which reflects this state by passing it to the rendered simulation. The simulation reflects this received snapshot by updating its internal entities. These entities are then rendered

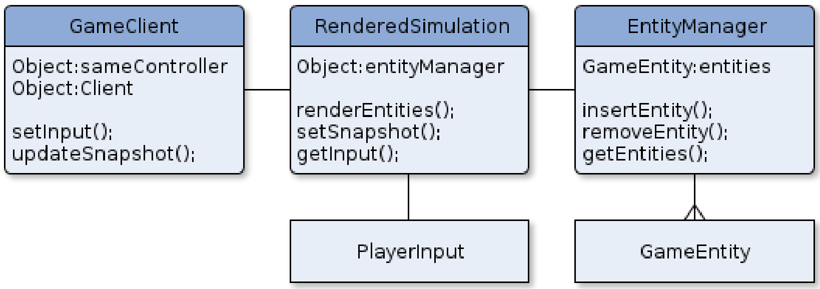


Figure 4.3: A class diagram displaying the internal relationships in the rendered simulation.

on screen in the game loop. The structure of a game entity is described in the following section.

4.4.1 Game Entities

According to the game design and the interaction matrix in Table 4.1 a set of game entities are to be defined. As described in this matrix these entities need to have different behaviour, functionality and should be rendered accordingly. A marble for instance has different physical properties and moves differently than a static wall.

A well known method for achieving modularized behaviour is the use of the composite pattern[2]. This pattern is adhered to by creating a group of well defined **components**. These components are interfaces that all sub components realize. By creating interfaces we ensure that any given component realize specific behaviour.

The **sub components** are responsible for the implementation of the actual logic. A sub component implementing the Renderer component knows that it should render *something*. How this rendering is achieved is up to the sub component.

The **composite** (also referred to as an entity) is the glue that binds these components together. The behaviour of a composite is entirely defined by its attached components. An empty composite would simply be a skeleton object.

The reason this pattern is favorable over the more traditional "behaviour by inheritance" pattern is that a small set of components can create the basis for a wide range of different entities. By using behaviour by inheritance instead a wide range of classes and class hierarchies would be needed to achieve the same goal. Figure 4.4 illustrates the composition of an entity (a wall in this specific example). The notation in this figure is Unified Modeling Language (UML) version 2.0 OMG as described by Martin Fowler[5]. This notation will be used throughout the thesis.

[2]Composite Pattern - `http://en.wikipedia.org/wiki/Composite_pattern`

The wall entity is comprised of three components. A renderer, physics and a contact response. All components share a small set of base functionality, which is why they extend the abstract class Component. Often two levels of inheritance will be sufficient to define a component. Two components will in most cases be sufficient to give an entity some meaningful behaviour.

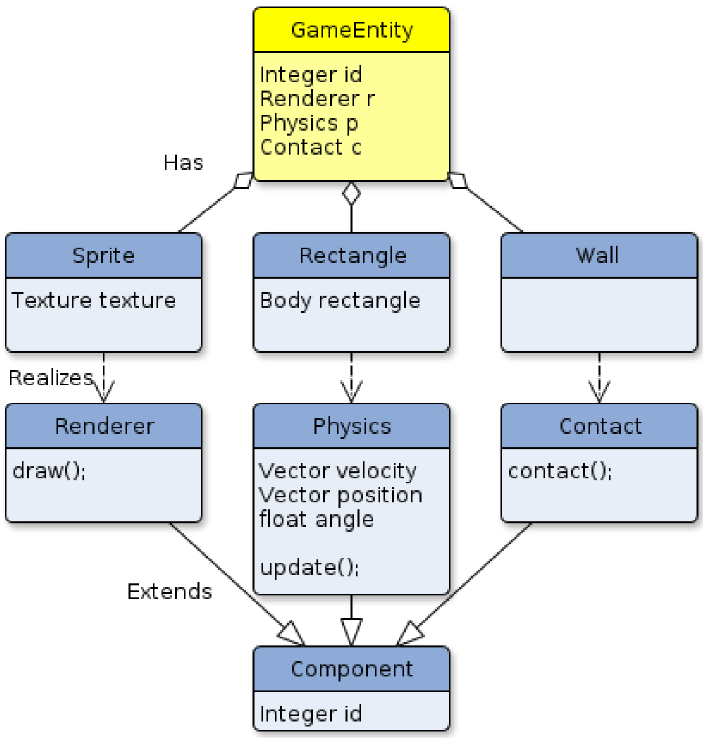


Figure 4.4: A game entity comprised of three components. These components are realizing behaviour from different abstract component.

The renderer describes how this entity should be displayed on screen. The attached component draws a sprite. Other renderers could render particles, animations or three dimensional models.

The physics component describes how this entity should be handled in the physical simulation. A wall for instance should not be movable and must have the physical properties of a rectangle. The marble should be movable, react to forces when colliding with a wall and have the physical properties of a circle.

The last component in this example is contact. This component describes what happens when some other entity touches this entity. A player controlled marble would be reflected in the opposite direction. Other entities such as a trap would have another contact response, which could inflict damage to the marble.

This design is adapted by several game engines and games. Nilson et al.[8] analyzed the game engine Torque[3], and decomposed that engine into specific sub modules. The identified components included:

- Input
- Physics
- Renderer
- Core

These components and components inheriting their behaviour is in many cases enough to describe game specific implementations in real physics based games. This will also be the case for the prototype where a simple yet representational implementation will be emphasized.

4.4.2 EntityManager

The *EntityManager* should be the only interface to add, remove and retrieve entities. Each entity is constructed by one or several components. These components have to be initialized and loaded with the appropriate data. Certain kinds of components have to be loaded in the game loop. A new entity however may be received from the headless simulation at random times. The manager should queue new entities and upon the game loop load and properly initialize these entities. An equivalent of this manager will run at the headless simulation. The difference is that the headless simulation should not load the rendering components, but only representations of these components. A rendering component could render some specific texture, and the component needs to contain this information even though it does not explicitly use it.

4.5 Network Architecture

This section will describe how the network architecture is to be implemented in this prototype. This includes how to distribute the game state, how this state is serialized and how errors in the network are handled.

Figure 4.5 illustrates the conceptual class diagram of the headless simulation. This is in several ways different from the diagram displayed for a client (see Section 4.4). The simulation should generate a list of connected players, and reflect their input in its entities. It should run entities through a *Physics* module and

[3]Torque - `http://en.wikipedia.org/wiki/Torque_Game_Engine`

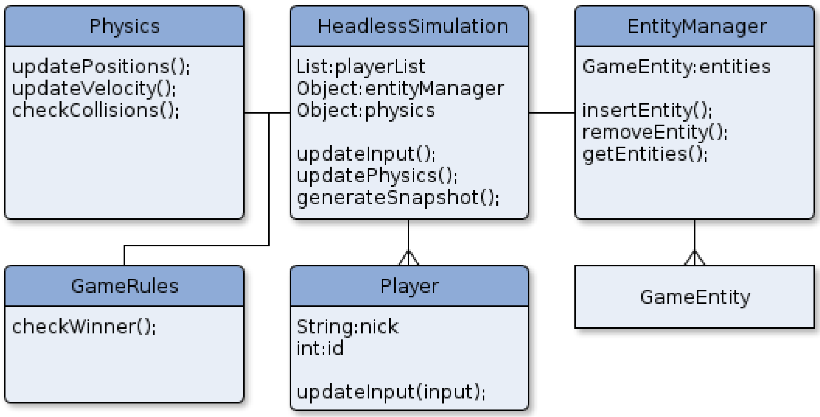


Figure 4.5: A class diagram displaying the internal relationships in the headless simulation.

check the game rules (if one player has won). The simulation should also generate a world snapshot and distribute this to the connected clients. This is further discussed in the following sections.

4.5.1 Passing Game State

Passing data over the wire is an expensive operation, and is prone to delays, errors and data loss. In multiplayer games a game state needs to be reflected at all participating players. In large fast paced games this state is often too big to be transmitted in its entirety every server tick. Bettner et al.[1] describes how they were able to synchronize game states with more than 1500 entities in the RTS game Age of Empires. This game had a *communication turn* of 200 milliseconds, which is slow compared to modern real time games. They could achieve this kind of synchronization by doing a few but important assumptions:

- All players are present at the start of the game
- All players are present throughout the game

By doing these assumptions the game could pass along the input from the players, and simulate the game at each client, such that all clients were kept synchronized. Once a player disconnected, the game got paused and saved. The game could not continue until all players were present again. One large issue with this system was that different implementations of floating point arithmetic in the processors, would lead to different computation results. Over time this difference accumulated and one client could have a very different perspective on the state than the other clients. As for most modern real time multiplayer games these assumptions cannot be made. Players can usually drop in/out at random times, and it's expected of a game to continue even though one of the players have disconnected.

One of the pinnacles of real-time multiplayer games the past decade was the Frames Per Second (FPS) game Counter-Strike[4] (see Section 2.1). This was initially a user created modification for the popular game Half-Life. This modification later on got acquired by Valve (the creators of Half-Life). The network architecture in Counter-Strike was based of another Valve acquired game named Team Fortress. This is a very successful client-server architecture where connected clients play on a pre-loaded map. The server expects that the client has the same version of the specific map stored locally. An important part in this architecture is to limit the amount of data that is sent between the client and the server. Valve argues that only objects that have changed should be transmitted and updated at the clients[11]. This methodology works great for games with a lot of static objects (which a level is comprised of), and some movable entities (players for instance). This would be the case for the game design as described in Section 4.1. Valve achieves this by creating *delta snapshots*. Instead of transmitting the full game state (a game snapshot) every tick the server iterates its internal game objects and generates a list of all the *changed* objects. These are transmitted, and updated at the clients. A prerequisite to delta snapshots is that clients originally must have the full game state. It would not give any meaning to update a lot of entities that does not exist.

A similar system will be implemented in the prototype. When a new client connects to a network, the master generates a full snapshot and passes this to the client. Once the client has generated his local version of the game world, the master starts passing delta snapshots to this client as well. A benefit by being able to create both "delta" and "full" snapshots is that the client does not necessarily need to have a locally stored level file. The server can generate this upon request and pass it to the client. The simulation must loop trough all entities in the game world, and check whether they should be transmitted. Since all the entities are comprised of components, it makes sense to let each component decide whether it needs to be transmitted or not. A moving physical component should always transmit its new coordinates, as opposed to a renderer component which only should transmit its colour once if the colour changed. This is realized in the component system by letting all components have a flag which indicates if that component has been changed. Upon a synchronization, the master will loop through its internal entities, and for each entity let that entity check whether it has changed components. If one or more of its components are changed it will append the data into a component snapshot. After the delta snapshot has been fully built it will be transmitted to all connected clients.

4.5.2 Serializing and Synchronization

As mentioned in Section 2.3 Same provides the master selection routine, and a protocol for sharing data between connected clients. This protocol is implemented

[4]Counter-Strike - `http://en.wikipedia.org/wiki/Counter-Strike`

as a concept described as *shared Variables*. These variables is how the programmer communicates with the back end. A shared Variable is denoted by a user-supplied name (the identifier) and a data type. The data type tells the Variable what types of objects it should expect to handle. When clients are to communicate through Same they simply create one Variable each which has the same key and data type. A Variable supports several operations for reading and writing values to it. These operations are described below.

get(): returns the current value of the variable. This is set to the most recent value the client has seen. This value will not be updated unless update() explicitly have been called.

update(): updates the variable. When calling this method the user acknowledges the new value before overwriting it.

set(): tries to set the value to a new object. This method will only succeed given that the variable is updated to the newest state.

To get notified of events in a Variable the client must register a ChangeListener to it. This listener will get notified once a variable gets updated with a new value. These ChangeListeners will be used to sample both input data from clients and update data from the master. When a variable is updated at a client, the protocol serializes the object and puts it on a socket connected with the master. The master receives the string representation of the object, checks that it is the most up-to-date data and further distributes it to all its clients. Once a client gets notified of an update to a variable it de-serializes the string and notifies its listeners with the object representation.

In the prototype these variables will be used to distribute the game state to all clients, and to notify the master of input. Hence two variables must be created. These are named:

- WorldSnapshot
- PlayerInput

The WorldSnapshot variable contains the simulation state in the master. The master runs through its entities and appends the changed entities to a list. Once this process is finished it updates the variable with this data. The clients should regard this as a *read only* variable. The value of this variable should only be set in the headless simulation context. All clients should attach listeners for this variable in their GameClient class. Pseudo code describing how the WorldSnapshot variable is set is given in Example 4.1.

Example code 4.1: Setting the value of the WorldSnapshot variable

```
// Headless simulation loop
while(running) {
    // Update entities
    world.update();
    // Update variable to latest state
    variable.update();
    // Set new state
    variable.set(world.getSnapshot());
    // Do sleep
    thread.sleep();
}
```

The rendered simulation in a client is updated when a change happens to the variable, as described in example 4.2.

Example code 4.2: The client gets notified of a world change

```
Variable<WorldSnapshot> world;
public void valueChanged(Variable<WorldSnapshot> unused) {
    // Got notified of change, update the variable
    world.update();
    // Reflect the new state
    simulation.updateWorld(world.get());
}
```

The clients interact with the world by passing their input to the headless simulation at the master. All clients set the PlayerInput variable, and they regard this as *write only* as shown in example 4.3.

Example code 4.3: The client transmits its input

```
Variable<PlayerInput> input;
public void setInput(PlayerInput newInput) {
    // Update variable to latest state
    input.update();
    // Set new state
    input.set(newInput);
}
```

The headless simulation has registered a ChangeListener to the PlayerInput variable and will get notified when any player sets this. Next the simulation should handle this input on the behalf of the client which set it. Pseudo code illustrating this is displayed in example 4.4.

Example code 4.4: The headless simulation gets notified of an input change

```
Variable<PlayerInput> input;
public void valueChanged(Variable<PlayerInput> unused) {
    // Got notified of change, update the variable
    input.update();
    // Assign input to a player
    PlayerInput received = input.get();
    // Do action based on input
    updatePlayer(received);
}
```

4.5.3 Master Selection

If the master gets interrupted or disconnected from the network, Same executes the Paxos routine and chooses a new master. Same provides a clean interface for listening on the state of the connection. This interface gets triggered when the connection state changes to either:

- Stable
- Unstable
- Disconnected

All clients attach a listener to this interface and they will immediately get notified of changes in the network. When a client hosts a new game Same will select the new client as master and that client will instantiate the headless simulation. This flow is displayed in the sequence diagram in Figure 4.6.

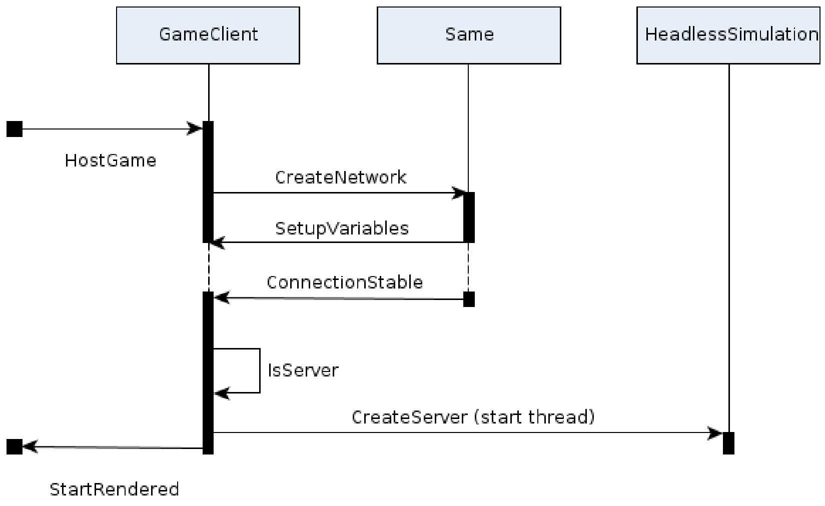


Figure 4.6: The sequence when a client hosts a new game. After the connection to "the network" (itself) is stable, the client will start to run the server simulation.

This figure shows the sequence that happens when a player presses "host game" on his device. The *GameClient* will create a new Same network, and setup the initial variables. This will trigger the master selection routine in Same, and later on give a connection state callback to the *GameClient*. If this callback is *STABLE*, *GameClient* checks whether it should create a server. This is checked by comparing the local IP address to the master IP address. If the client was chosen as master it starts a new *HeadlessServer* thread.

The use of the connection state callback from Same to create a server greatly simplifies the implementation for when connections are lost or the master gets disconnected. The callbacks will tell what client to create a server. The equivalent sequence diagram for a client connecting to an existing game is shown in Figure 4.7.

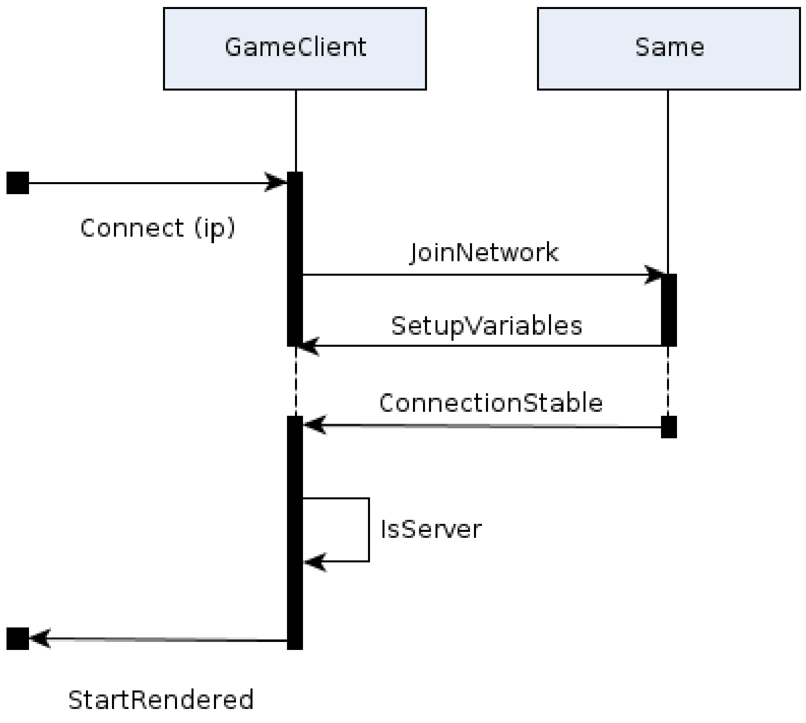


Figure 4.7: The sequence when a client connects to an existing game.

This sequence is different from hosting a game, since the client provides an existing IP address to connect to. After triggering *JoinNetwork* in Same, the client must wait for the connection state callback. If this callback returns *STABLE* the *isServer()* check will fail (as expected). The *RenderedSimulation* will start and request a full game state from the master.

4.5.4 Failures and State Recovery

There are two main cases of failures that may happen within this system:

1. An arbitrary client may fail
2. The master may fail

The first case will get handled by the failing client itself. If it was an intermediate failure, it will simply reconnect to the network and request an update. If it was a longer outage the user may reconnect to the game at a later time, as shown in Figure 4.7.

The second case is more cumbersome. Since the master is authoritative the clients will not know what is the correct game state once the master fails. This problem is partially solved by Same. Part of the master selection routine inspects what client updated the variable last, and accepts this as the "newest data". This client is a good candidate for becoming the new master. Since the data set at this time is in a unknown state (every client got their own version), the system have to settle with the latest state from one of the clients. The master failure and selection of a new client is shown in Figure 4.8.

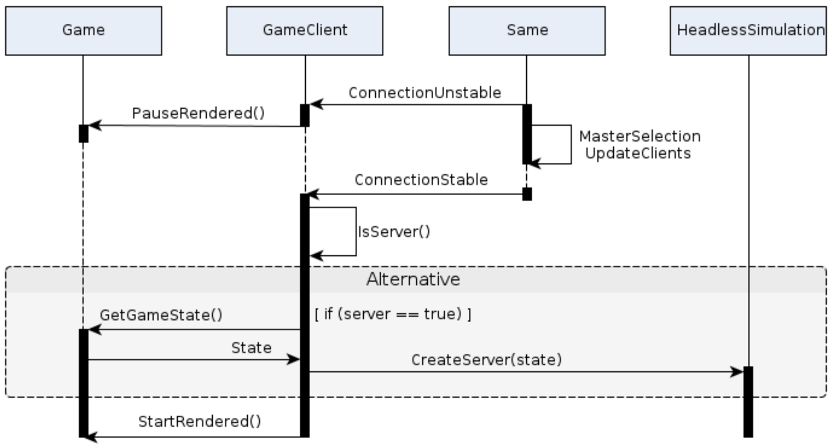


Figure 4.8: The sequence when a master disconnects, and this client is chosen as a new master.

It is crucial to pause the rendered simulation if connection to the master is lost. This is done immediately after the clients receive the *UNSTABLE* connection callback. The longer a client runs its own simulation, the more the game state between the clients will differentiate.

Same runs the master selection routine in the background, selects a client as the new master and eventually gives a *STABLE* connection state callback. Upon receiving this callback every client check whether they became the new master. If this is the case for a client, it serializes its own rendered simulation data set, and passes this as an argument into the newly created headless simulation. The headless simulation picks up where that client last got paused, and starts updating the other clients. All clients resume their local simulation and get updated according to the state from the new master.

Chapter 5

Implementation

The goal of this chapter is to describe the implementation phase. This includes libraries extended and specifically why these libraries were chosen. This chapter will also describe methods applied in order to achieve an effective state replication. Problems encountered and proposed solutions to these problems will also be discussed.

5.1 Libraries

This section will describe libraries and frameworks used to develop the prototype and emphasize why these frameworks were chosen. Common for these libraries are that they are written in Java[1] or have an equivalent Java port. They are also open source software.

5.1.1 Same

What Same is and what it aims to achieve in this prototype has already been explained in Section 2.3 and Section 2.4. This section will give a description on how Same was adapted and implemented in the prototype. The author of Same provided a thorough example application which helped a lot in the initial adaption of Same.

[1]Java - `http://java.com/en/`

Implementation:

Same was added to the prototype project in Eclipse as a Maven dependency[2]. Maven is used to make it simpler to handle a large set of Java dependencies. These dependencies can be added through an online repository or through a local repository. The Same framework was published as a Maven Artifact on Github which simplified the process of adding this project as a dependency.

A *GameClient* class was created, which holds references to two of the core classes in Same: *SameController* and *Client*.

SameController: This controller is the main interface to the Same back end. The controller is used to create networks, instantiate the local client and to add the ConnectionState-listener.

Client: This class is used to join specified networks and to create shared variables in the context of those networks.

The *GameClient* class encapsulates the two aforementioned objects, and abstracts their functionality into two concise methods as described below:

host(): Hosts a new game. If this method is triggered, the GameClient will use the SameController to create a new Client and then create a new network. The IP for this network will be one of the IPv4 addresses in the network interfaces for this device.

connect(String IP): Triggered when the user wants to connect to an existing game. This method uses the SameController to create a new Client and then connects to the existing network through the Client. The IPv4 address of the current master must be passed as an argument to this method.

These two methods share a lot of functionality (creating the *SameController* and retrieving the *Client*) hence a general private method *setupClient()* was created. This method does the initial setup of the *SameController* and the *Client* regardless of this device hosting a new game or joining an existing one.

5.1.2 LibGDX

Creating a game from scratch could be an enduring task but there exists countless game frameworks that try to simplify this process. The main goal of these frameworks is to provide the tools for making games instead of having the developer focus their time on making these tools. It is important to consider whether it is worth spending time to learn the framework as opposed to tailor a game from scratch. The developer on this project had some prior experience with a cross platform library named LibGDX[3] which was the main reason this framework was

[2]Maven - `http://maven.apache.org/`
[3]LibGDX - `http://libgdx.com`

chosen. This is a popular open source game library written in Java. LibGDX enables and makes it simple to deploy to several platforms including Android, Desktop and Google Web Toolkit (GWT) based on a shared code base.

This library provides a game loop, an Open Graphics Library (OpenGL)[4] context, a wide range of modules, ranging from a scene graph, math utilities and OpenGL abstractions. Most of the modules in this library are optional meaning that the overhead by using the library is small compared to complete game engines. Merely the skeleton of a game engine is provided by LibGDX and it is entirely up to the developer on how to implement the engine.

Advantages by using LibGDX:

- Quick prototyping. Abstractions for low level data handling such as file input/output and rendering data on screen.
- A bare bone setup for creating the game loop (a class implementing the ApplicationListener)
- Cross platform, meaning that it deploys to both Android and desktop. It is more time efficient to do prototyping in a desktop environment, instead of launching the project on a device or the Android emulator.

Disadvantages by using a LibGDX:

- Bugs in the library can indirectly affect the implementation of the prototype.
- Ties the architecture of the prototype to an already existing architecture in the library (MVC based architecture).

One of the main concerns in this project was to create a prototype on top of Same during a short period of time. The quick prototyping enabled by the use of this framework combined with the developer having prior experience with it was the main reason LibGDX was chosen.

Implementation:

The library was checked out from Github[5] and imported into Eclipse as an existing project. In Eclipse the library was referenced to by the prototype as a source library. This made debugging and source look-up simple. The next step was to implement the *ApplicationListener* from LibGDX. This is an interface providing methods for the life cycle of a LibGDX application. The methods implemented are described below:

create(): Called once when the application is started. This method should be used to create important application objects.

resize(int width, int height): Called every time the game screen is resized.

[4]OpenGL - `http://en.wikipedia.org/wiki/OpenGL`
[5]Libgdx (Github) - `https://github.com/libgdx/libgdx`

render(): Called by the game loop every time rendering should be performed. This loop should also update the game logic.

pause(): On Android this method is called when the Home buttons is pressed, or a phone call is incoming.

resume(): Called on Android when the application resumes from a paused state.

The *create()*-method instantiated *GameClient* and *RenderedSimulation*. This method also loads assets (textures and fonts) which are used in the prototype.

The *render()*-method was used to update game logic, the simulation and to act on data received from Same. The *render()*-method tries to update the screen at most 60 times per second. For graphics running on regular computers this is often the case. The same assumption cannot be made in regards to Android devices. Modern devices have rather powerful Graphic Processing Unit (GPU) chips but some of the older devices tend to suffer when trying to render graphic intensive applications. The vast amount of hardware configuration is one major issue when creating games for the Android platform.

5.1.3 Box2D

Architectural requirement #5 (see Section 4.2) states that the simulation is to be based on mechanical physics. Primitive physics was added in the early stages of the implementation. This included movement, gravity and collision detection between simple shapes (circles and rectangles). As the development progressed so did the desire for more advanced physics. Physics including forces and responses to collisions would make the gameplay more enjoyable. Box2D[6] is a widely used and mature physics engine for two dimensional games. Natively it was developed in C++ but there exists ports for several other languages including Java. After looking into the documentation, replacing the self made physics with Box2D seemed feasible. However the use of Box2D entails more complexity in the architecture. A *World* object has to be instantiated. The *World* simulates, performs collision detection and collision responses between all *bodies* contained in it.

Implementation:

The Box2D *World* is the physics hub that manages memory, objects and simulation. Therefore a *World* was instanced in the headless simulation to run the simulation of physical entities. The clients should only reflect this simulation. As Section 4.4.1 describes each entity is comprised of one or several components. The generic component *Physics* was extended to a special component *Box2DPhysics*. This component references a *Body*. Only a few primitive shapes are available in this component (rectangle and circle). The game loop of the headless simulation progresses

[6]Box2D - `http://box2d.org/`

the physical world by triggering the worldUpdate()-method of the *World* with a fixed time step.

Box2D provides collision detection in a two dimensional world space (X and Y). After experimenting some with this library it became apparent that adding a third component Z (depth) was possible. This was implemented as a proof of concept, and allowed for three dimensional movement and collision detection. Essentially this was achieved by letting Box2D do collisions in the X and Y space, and if a collision was detected (two shapes overlaps in the two dimensional space), a depth test was carried out to test whether they collide in the third dimension. Adding an additional degree of freedom meant that more values have to be transmitted to the clients, since entities can move and be positioned in a Z space as well.

5.2 Entity Replication

The entity replication was initially implemented by having two shared variables for replication, namely the FullWorld and DeltaWorld variables. The FullWorld variable was used for distributing a full state of the game world, whilst the Delta-World only distributed delta state. The master would continuously write to the DeltaWorld-variable, while the FullWorld-variable only was written to when new client appeared or a new in game entity was created.

There were several problems with this solution. The most severe problem was that when a new entity arrived, the client could receive a delta snapshot while creating the new entity. This often resulted in a crash, since the delta snapshot tried to update components that did not exist. This was avoided by using a mutex[7] to only allow delta updates if the game world at a client was fully constructed. However this led to other issues and more complexity. The prototype had to check several places if the world was currently rebuilding and the client could discard several delta updates as a consequence of this.

As Section 4.4.1 describes, each game entity is given an internal identifier. When a snapshot is created for an entity, the snapshot is signed with this entity identifier. When this snapshot is received at a client the client updates its local version of that entity based on the identifier set in the snapshot.

By adding a flag to the entity snapshot indicating whether it is a full or delta snapshot the issue with two variables and partially snapshots was avoided. When a new entity was created at the master, the flag is set to true. Once the entity has been fully transmitted to the clients the flag is set to false and subsequently only delta snapshots of that entity is being transmitted.

When the client processes a world snapshot, it loops through the entity snapshots and for each snapshot checks whether that identifier already is present in the simulation. If this is the case the client updates the entity, if not it checks whether

[7]Mutex(lock) - `http://en.wikipedia.org/wiki/Lock_(computer_science)`

the full flag is set. If the flag is set the entity is created. If the flag is not set, this is a delta snapshot for an entity that is to be created or has been removed. Pseudo code for this approach is displayed in example 5.1. By using this approach the FullWorld variable got discarded. Instead of letting what variable a snapshot propagated through decide whether this was a new entity this was decided by the flag in the entity snapshot.

Example code 5.1: Entity update at a client

```
// A single EntitySnapshot is handled
private void updateEntity(EntitySnapshot snapshot) {
    // Check existing
    GameEntity existing = manager.get(snapshot.id);
    if(existing !=null) {
        // Identifier exists, update
        existing.update(snapshot);
    } else if(snapshot.full) {
        // Append a new entity into the engine
        createEntity(snapshot);
    }
}
```

One problem with this method was snapshots received out of order or not received at all. This approach naively assumes that a received snapshot is the next one that should be processed. If a full snapshot is missed by a client it will not be able to create that entity, and will continuously discard any delta snapshots received for that entity. If that client is selected as master, it will distribute its known entities which could lead to a game world inconsistency.

Svein Erik suggested a method for having more control over the life cycle of entities. Instead of having a boolean that indicates whether the snapshot is full or not, the snapshot could have a state identifier. This identifier should continuously be updated with the state received from the master. A new entity would have the identifier 0. When a client receives this it locally creates the entity. This would also make it possible to update entities in-order. When the master serializes its game world it increments all state identifiers. A client could update its local world in the correct order by comparing its local entity state identifier with the one in the received snapshot.

This method would also enable a client to detect missing snapshots. If the client receives a snapshot with an identifier offset larger than 1 it knows that the snapshot was received out of order. The client could request a retransmit from the master. All clients would see this retransmit, but only the clients who "are behind" would update their worlds. This mechanism would also require the master to cache its transmitted snapshot for a time. Implementing this feature is left to further work.

5.3 Handling Updates and Synchronization

In this prototype clients run their game loops asynchronously. The master tries to output data at a steady rate, but due to delays in the network and the master itself, this rate will often vary. The clients have no control over when they might receive an update from the master.

This became a problem early on since the clients naively processed received world updates directly after receiving them. Most of the time the update was processed successfully but in some cases the application would crash due to instantiation of a physical body while the physical simulation progressed or when loading a texture without the OpenGL context being ready. This was traced down to concurrency issues. Each Variable in Same is running in its own thread to avoid blocking the parent application. Likewise the game loop runs in the main Java thread. If an update was received when the main game thread was at a critical area of the game iteration (calculating collisions in the physical world) the application would crash with a concurrent modification (or similar) exception.

This issue has nothing to do with Same specifically, but rather how a multi threaded application should be designed. The solution chosen for this prototype was to append every world update to an atomic list at the client. When the client runs its game loop it processes and removes every item in this list and then continues to update its local world. The same approach was applied in the master when it receives input from the clients. Each issued input command gets appended to a list and when the master runs its game loop it pulls these commands from the list and processes them accordingly. These lists were implemented by using the *AtomicQueue* in Java. By using a normal list or queue, threading would still be an issue since the main game loop might iterate the list when the variable gets updated with a new element which would populate an exception.

5.4 Issues by using Same as a platform

This section will describe different issues encountered when applying Same as a multiplayer game platform. The section will also propose solutions on how to potentially fix these issues.

5.4.1 Client Differentiation

In a multiplayer game it is often necessary to know what client executed a given action. In most client-server architectures this is achieved by having the server keep an internal state of all connected clients and assign an identifier to the client connection. When a client pass input to the server the internal player state is retrieved for the connection identifier and the server executes input by using the player state.

A challenge encountered early on in the implementation phase was to differentiate between clients. Same has a notion of all clients that are connected to a network, but no notion of which client triggered the update of a Variable. In early stages of the game this was naively implemented by letting clients attach their location (IP address) when setting the input Variable. When the master received this input it parsed the location and updated the game state accordingly.

This was merely a workaround, and could lead to several problems:

- A mischievous client could set the IP address of another client, making it act on the behalf of that player.
- If the IP address of a client changes, it is essentially registered as a new player (unlikely case).
- Dependent on clients adding their IP address, could be prone to bugs in different software version.
- More data overhead for the input variable. Every client need to update the input variable with their location.

A more robust solution would be to let the master in Same internally broadcast where changes originated from. Same is conveniently distributed as an open source project. This made it possible to decompose the framework and add the desired feature. This was achieved by adding another optional field *location* to the *Component* class in Same. The Variable classes in Same are abstraction layers on top of *Component* (not to be confused with the components of an entity). The location field denotes the location of whom updated the variable (the client set a new value in the component). When the master receives an update, it attaches the client location and broadcasts this to all listening clients.

How Same initially broadcast changes in variables to clients is displayed in example 5.2.

Example code 5.2: The regular Same interface

```
// A variable was changed
public void valueChanged(Variable<Type> variable) {
    // Do stuff
}
```

After implementing the changes as discussed above, one argument was added to the interface as shown in example 5.3.

Example code 5.3: The interface with location

```
// A variable was changed
public void valueChanged(Variable<Type> variable,
    String location) {
    // Do stuff based on location
}
```

When a Variable change is notified to the listeners the location is also passed as an argument. This enables the client acting as server to update a player according to what client issued the update.

5.4.2 Jackson Serialization

As mentioned in Section 4.5.2 internally Same uses a library named Jackson for JSON serializing. When a Variable is updated with a new object, this object gets serialized into a JSON string and passed to the master. Once a client gets notified of a change in the Variable the received data gets de-serialized and a Java object is passed to the ChangeListener.

As described in Section 5.2 different data types are used to represent the properties of an entity snapshot. One of the more complex data types is the Vector class, which is part of a mathematics package in the game framework. This class is comprised of three fields (X, Y and Z), and has a wide range of methods for doing vector operations. The problem with this class is that it's not Jackson compatible for serialization.

In order for Jackson to properly serialize fields in a class, those fields must either:

- Be declared as Public
- Have getters and setters defined
- Be annotated with @JsonProperty("fieldName")

Since the vector class is part of another library implementing this change is non-trivial. One solution would be to extend the vector class, and use the extended class for serialization instead. Another solution would be to ask the maintainers of the library to add that change. In most cases these solutions are not feasible. The library could be closed source, or the maintainers could deny/not respond to the request for *some* reason.

The solution chosen during the implementation phase was to do the object serialization with another JSON library, before setting the Variable. When the Variable gets set, Jackson detects that it already is a String and transmits the data. The other JSON library was part of the game framework, and did not have the same field requirements as Jackson. This workaround could be properly implemented by creating a generic serialization interface as shown in example 5.4.

Example code 5.4: Generic JSON interface

```
public interface JsonInterface<T> {
    // Writes the object as a String
    public String writeValueAsString(T value);
    // Reads the data into an object
    public T readValue(String data);
}
```

When a Variable gets created this interface could be supplied as an argument and the Variable will be serialized using the interface. How the actual serialization is carried out depends on how the developer has implemented the two methods. This is not limited to JSON as long as the data gets serialized to a String and can be deserialized back to an object. The interface must also be set in the *State* object in Same.

5.5 Delta Snapshots

As explained in Section 4.5.1 the game state should be propagated by using full and delta snapshots. A pure data class *Snapshot* was created for this purpose. This class was designed so that it has sufficient fields to suit all created components in the prototype. A fill(Snapshot snapshot)-method and a *getSnapshot()*-method was implemented on component level. Every class extending component had to realize these methods. The fill()-method would be triggered at the client and update the client state of that component. The *getSnapshot()*-method would be triggered at the master and would fill and retrieve the snapshot for this particular component. Attached in Example 5.5 is small excerpt of this generic snapshot class.

Example code 5.5: Generic snapshot class

```
public class Snapshot {
    // Id of this component
    public int id;
    // Float (rotation)
    public Mutable.Float f_0;
    // String
    public String s_0;
    // Position, velocity
    public Vector3 v3_0;
    // Tint for
    public Color c_0;
    // More data types ..
    public Snapshot() {}
}
```

When a component is to retrieve a snapshot it fills out the corresponding fields in the snapshot from its internal component fields. Likewise when a snapshot is received the component updates its corresponding fields from the snapshot fields. The identifier of the snapshot is directly mapped to the component so that the client know what component to update. A snapshot does also have a method for checking whether it contains any data (hence should be transmitted) and to create a deep copy of itself. An example of a JSON encoded delta snapshot of a physical component is shown in Example 5.6. The first vector object denotes the position of the component and the second vector denotes the velocity of the component. The float value is the rotation of this physical component. Note that only set values gets serialized into JSON since fields equal to null or the numeric value 0 also will de-serialize to null or 0. This is why most JSON processors omits fields with these values when serializing.

Example code 5.6: JSON representation of a component snapshot

```
{
    v3_0: {
        x: 7.995022,
        y: 5.4424934,
        z: −3.0071216
    },
    v3_1: {
        y: −3.7025578,
        z: −0.84393203
    },
    f_0: {
        value: 39.120243
    },
    id: 6
}
```

5.6 Entity Interpolation

As discussed in Section 5.1.2 the prototype will at most render at 60 frames per second. This is the best case scenario and in a perfect world all clients in the system updates and renders at the same rate.

However this is not the case, and more often than not when dealing with mobile devices. «Tick-rate» is a term coined by Valve and is defined as one "logical" step at the server. This logical step includes processing received user commands, run physical steps (if any), check game rules and send a snapshot to the clients. This rate is tied to how fast the server can process one logical step. It can be assumed that a snapshot update may be received at any time. It can also be assumed that world updates will be received at most 60 times per second.

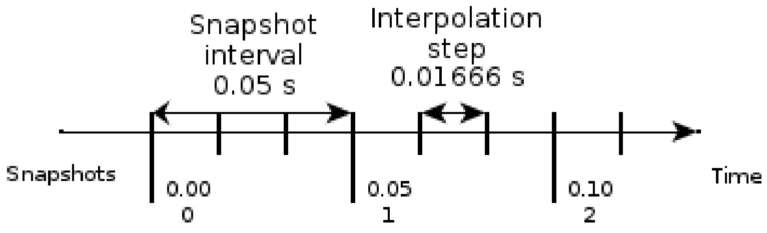


Figure 5.1: Snapshots are received at time 0.00, 0.05 and 0.10. This figure displays how a client interpolates between two received snapshots (tick 1 and tick 2)

If a client updates its screen at 60 frames per second and the server runs at 20 ticks per second, the client will experience jittery and unsteady gameplay. 2/3 of the draw frame calls will render the entities at the same position. Jittering becomes even more apparent if the updates are received at an unsteady rate.

To compensate for this Valve[11] suggest to interpolate entities between updates. In short this means that a client will cache one or several snapshots and use the interpolated values between two snapshots to represent a correct world image relatively to time. This means that a client will render behind the current state of the server. A conceptual illustration of this interpolation is given in Figure 5.1. The example assumes that snapshots are received 20 times per second (every $1/20 = 0.05$ second). And that the client simulates its local world 60 times per second (every $1/60 = 0.0166667$ second).

Interpolation in the prototype was implemented on a component level, more specifically the *Box2DPhysics* component. It is not necessary to interpolate components that are limited to discrete values (a renderer changing the colour of an entity), but it is desired to interpolate values that change between each received update (such as position and velocity). A few noticeable changes had to be made in the *Box2DPhysics* component. The two last received snapshots had to be cached. This was done in the *fill()*-method which gets triggered upon each received snapshot. The interpolation itself was implemented in the *runClient()*-method, which gets triggered each frame draw in the client. A simplified example of interpolation over position is shown in Example 5.7.

If the interpolate flag is set, the current progress between two snapshots will be calculated. This is dependent on the tick rate of the server. The three position components will get set to the current (latest) snapshot received. If this iteration is in between two snapshots position will be interpolated between those values based on how this far simulation locally have progressed. Next the local position will be updated to reflect the interpolated position. Last the accumulated time in between two snapshots will be appended.

Example code 5.7: Position interpolation

```
public void runClient(GameEntity entity, float delta) {
    // Flag for interpolation
    if(interpolate) {
        // Current progress between two snapshots
        float progress = accumulated/tickRate;

        // Assign from last snapshot
        float x = current.x;
        float y = current.y;
        float z = current.z;
        if(progress < 1f) {
            // Do interpolation
            x = interpolate(previous.x, x, progress);
            y = interpolate(previous.y, y, progress);
            z = interpolate(previous.z, z, progress);
        }
        // Update body
        updateTransform(x, y, z);

        accumulated += delta;
    }
}
```

5.7 One Simulation

As the initial design suggested (see Section 4.3), the master should run two concurrent simulations. The headless simulation being responsible for the game logic and physics computations, and a rendered simulation rendering all entities on screen. The clients on the other hand should only run the rendered simulation and draw the game state state according to the master.

The prototype was initially implemented using this design. However the performance of older devices (Nexus One) suffered when they ran as master. This was due to several reasons:

1. Serialize the game world in the server thread only to de-serialize it again in the renderer thread.
2. Propagating the serialized world to all clients (setting the WorldUpdate Variable).
3. Having to simulate two physical worlds (one in the rendered simulation and one in the headless simulation) added additional performance overhead.

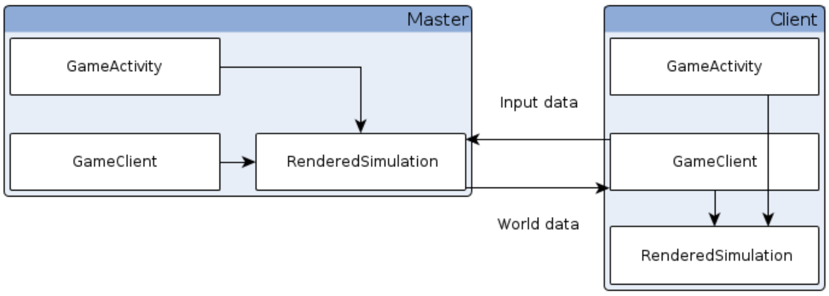


Figure 5.2: A master and a client communicating after the architecture was simplified to one simulation.

As a measure to make the prototype perform faster on older devices it was decided to remove the headless simulation. One of the largest gains by having a headless simulation is that the server software can be deployed on devices whose only task is to simulate the game logic (a dedicated server). Since every client in the Same network is supposed to render the game world this benefit does not outweigh the performance issues.

Another disadvantage by having two simulations was the complexity of starting and stopping the headless simulation. When a client was selected as a master (see Figure 4.8) it had to serialize its game world and feed this as an argument to the new thread. The thread had to be started and the game world had to be rebuilt. The thread would also have to create and attach listeners to the shared Input and World-variables.

After deciding to simplify the implementation to a single simulation a few noticeable changes had to be made. All clients instantiate one simulation object which is a hybrid between the headless and rendered simulation. This simulation contains the game world and references to the variables created in the *GameClient* object. A simple check in this simulation object dictates whether the client is acting as a server or a client. When an arbitrary client gets selected as master, instead of firing off a new thread it will start updating the game world and set the World-variable with its state. The master will also start to listen for changes on the Input variable. Clients will act as before listening for the world update and update their entities accordingly. Figure 5.2 displays the updated architecture.

5.8 Handling New Players and Disconnects

To let a client control a specific game entity it is necessary to keep a mapping between a client and the game entity. This was achieved in the prototype by letting the master maintain a list of connected players. Every time a client updated the input variable, the master would retrieve the correct player object (based on location). If the retrieved object was null, no player object existed and in this case the master would generate a new player object and put it in the player map hashed on the client location. Consequentially the master would generate and pass a full world state to all listening clients.

Next time the client sets the input Variable the master will retrieve the player object and update that client's game entity. A problem with this implementation occurred once the master failed. A new client got selected as master and dynamically built its player list when receiving client updates. This would generate a new entity for each player, even though there already existed entities for those players in the game world. The solution to this problem was to let the game entities have an "owner" field. The field would be the location of the client that originally generated the entity. Before a master creates a new player entity it will enumerate its entities and retrieve the game entity that has an owner field equal to the location of the client. If no such entity exists the master will create and append a new game entity to the simulation.

Chapter 6

Evaluation

This chapter aims to evaluate the results of this master thesis. The evaluation is based on the problem statement in Section 1.1. Since the framework is to be evaluated based on a multiplayer real-time game it is natural to look at the performance in Same and data throughput through the system. This rate is also coupled with the performance of the device running the prototype. Bugs and problems encountered when adapting Same will also be discussed.

6.1 Test Execution

The tests will be conducted on two different test cases. The difference in these cases are the amount of data that will be transmitted. This is achieved by creating two game levels with different amount of game entities to distribute. The two cases are:

Test case #1: 5 static game entities. These entities will make up for a platform that the marbles (players) land on. When all clients are connected this will total to 9 game entities simultaneously.

Test case #2: 20 static game entities. These entities will make up for a more advanced level, hence more state to distribute. This will be a total of 24 game objects.

Figure 6.1 is an in-game screen-shot from test case two. This level is comprised by 20 static blocks. Two players are currently connected to this game. The square platform to the left in the picture is the basic platform that is used in test case one. Each block is constructed by using a renderer component (the color and texture of the block) and a physical component (the size and position of the block). This is also the case for the player marble which in addition has an input component (dictates how input is handled) and a label component (the textual nickname representation). When a new marble is added to the game this marble is assigned a name (the nick of the player) and a random colour. Note that this level and its game entities differs some from the initial design proposed in Section 4.1.1. Instead of having traps and pits, the walls act as platforms which the players can jump to. If the marble rolls of this platform it will be destroyed and the player will have to re-spawn.

The number of updates per second issued by the master (tick-rate) will in these test cases be configured to 33 and 60. A higher tick-rate will demand more resources of the master but also lead to a more responsive client experience. These tests will be conducted for both delta snapshots enabled and disabled. The purpose of this is to see how Same copes with transmitting a larger set of data. When delta snapshots are disabled the full game state will be transmitted each tick.

There are four devices available for these tests, and both test cases will be run having each device acting as the master. The other devices will sequentially connect to the network. The available devices are:

- Nexus One (see Table A.1)
- Nexus S (see Table A.2)
- Nexus 4 (see Table A.3)
- Asus Laptop (see Table A.4)

The hardware and software specifications of these test devices are listed in Appendix A.

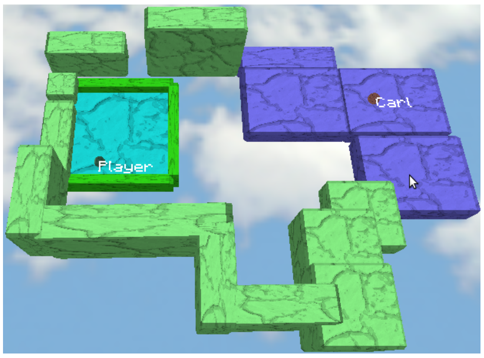


Figure 6.1: An in-game screen-shot from the prototype. This is the level that was used for test case #2.

The desired result from these tests is to observe how Same handles the continuous propagation of a larger game world to the clients and how several clients affect this propagation. Hence a set of metrics will be observed:

FPS: This will be a metric on how powerful a device is. A powerful device will render at 60 frames per second, a less powerful device on the other hand could render its screen at a lower rate.

Tick-rate: How many updates does the master manage to propagate? This is indirectly coupled with the FPS/performance of the master. This will be measured in ticks per second.

Data size: How much data is propagated through Same? This will be observed by data is sent through the socket which Same creates.

6.2 Tick-rate and Frames Per Second

It is assumed that there is a direct correlation between the tick-rate of the master and the master performance. The implementation of the server simulation can at most run one tick for each render call (the frame rate). This means there is an upper bound of 60 ticks per second, given that the master runs at 60 frames per second. A threshold has been implemented which makes it possible to limit the tick-rate independently of the frame rate. Even if no virtual limitation on tick-rate is implemented it is not given that updates issued to clients will run equivalently with the frame rate of the master. This is dependent on the internal mechanisms in Same, response times from the clients and delays in the network. The graphs in the following sections are based on sampled data from the prototype. This data is attached in Appendix B. The different test cases are described in Section 6.1. The goal of these sub sections is to display the correlation between the master performance and the rate at which it can propagate data through Same.

6.2.1 Test Case #1

Figure 6.2 shows the relationship between tick-rate and the frame rate of the master. This test case was run with an unbound tick-rate and by the use of delta snapshots. As assumed there is a clear correlation between the ticks issued by the master and the master performance. The plots for each device is plotted when that device is acting as the master.

This figure displays that there is a clear correlation between the master performance and how many ticks it manages to process each second. The two weakest devices in this comparison manages to keep a one to one ratio between render updates and tick updates. It is interesting to see that the Nexus 4 on the other hand has an upper limit for the tick-rate at 43 ticks per second, even though it renders the screen at about roughly 55 times per second. It can be assumed that the state propagation to some extent is Central Processing Unit (CPU) bound. The laptop manages to propagate at roughly 60 times per second. This device has a superior CPU compared to the other devices. It becomes clear that the tick-rate is directly affected by the number of clients connected (as was implied in the documentation for Same).

The same test was carried out when limiting the tick-rate of the master to 33 ticks per second. The main goal for limiting tick-rate is to lessen the CPU load on the device acting as server. These results rendered similar to Figure 6.2 except that the tick-rate upper bound started at 30 ticks per second for the stronger devices. There was not much gain in the frames per second for each device though. The simulation runs fine at 33 ticks per second. The clients get an adequate amount of updates and the local interpolation ensures that the gameplay is smooth enough.

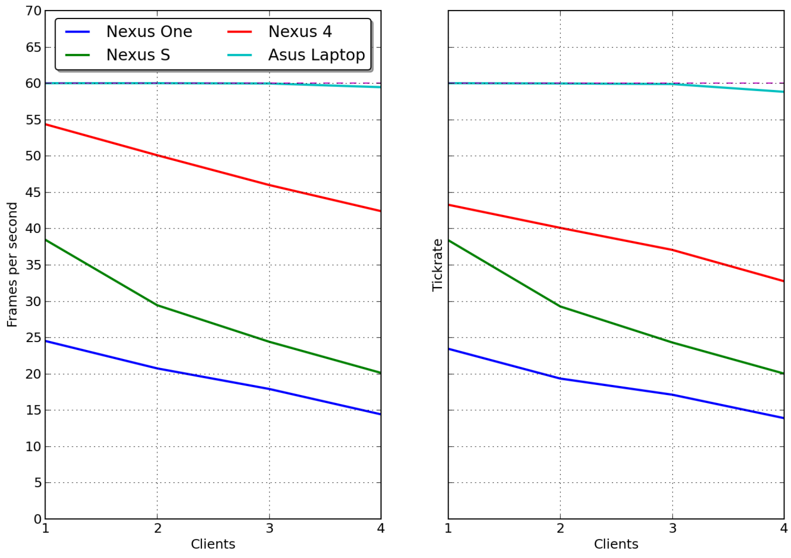


Figure 6.2: Average frames per second and tick-rate for test case #1. The plot is dependent on the number of connected clients. Each plot describes the performance of the device acting as master.

6.2.2 Test Case #2

The goal of test case #2 is to measure the difference in performance for the devices when propagating a significant larger game world. This test will also give an indication on how well the size of the game world will scale.

Figure 6.3 shows the average performance for test case two. This graph is considerably similar to Figure 6.2. One of the main differences is that the FPS generally has a lower starting point for all the devices. This is due to the fact that more entities are rendered on screen and more entities are part of the physics simulation. When additional clients join the game a close to linear decrease in performance is noticed. Common for the two cases is that the Nexus One device really struggles to keep up. 30-25 FPS is considered the minimum for a playable realtime game. The low performance is also noticeable in the connected clients which receive game state updates at a low rate.

When the devices are acting as pure clients the decrease in performance is not that apparent. The average FPS of the devices is also generally 15-25% higher (than when the device acts as master). This supports the assumption that there is a non-trivial impact on the performance of the device acting as master.

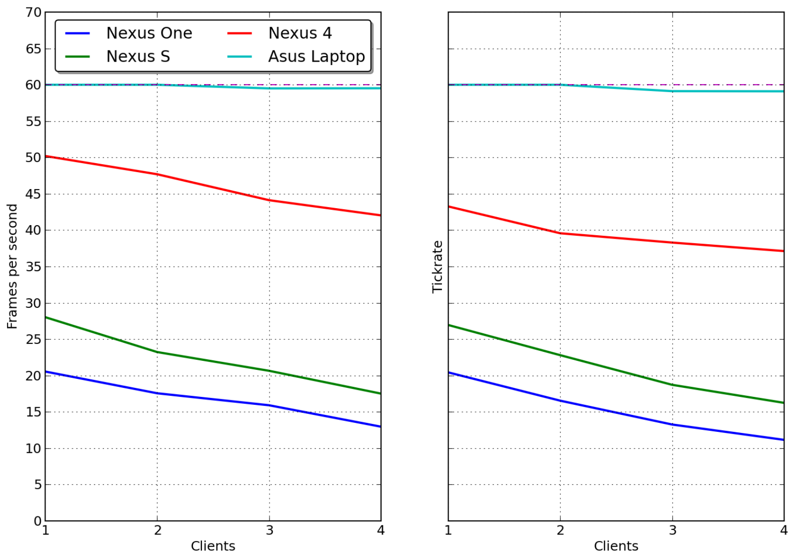


Figure 6.3: Average frames per second and tick-rate, dependent on the number of connected clients. Each plot describes the performance of the device acting as master.

6.3 Transmission Rate

It is interesting to test how much data Same is able to propagate. During the initial implementation it became obvious that there is a limit of how much data can be sent at a time. This could be limited by the wireless interface in the device, distance to the wireless access point and other external factors. Passing delta snapshots as explained in Section 5.5 was one method to avoid congesting the transmission channel. It is desired to measure the effect of delta snapshots. This data was measured through the desktop application of the prototype.

There exists several open source tools for monitoring bandwidth usage on a Linux system and *iftop*[1] for its simplicity. This tool gives detailed information about running sockets on the system. The Android SDK does provide a network debugging tool for application running on Android version 4.0.3 or later[2]. Only one of the test devices run this version though, and it was easier to monitor the network directly through the operating system.

To simplify testing this test was carried out by having the master run at 33 ticks per second. Data transmitted was only measured between the master and one single

[1]iftop - `http://en.wikipedia.org/wiki/Iftop`
[2]DDMS Network - `http://tools.android.com/recent/detailednetworkusageinddms`

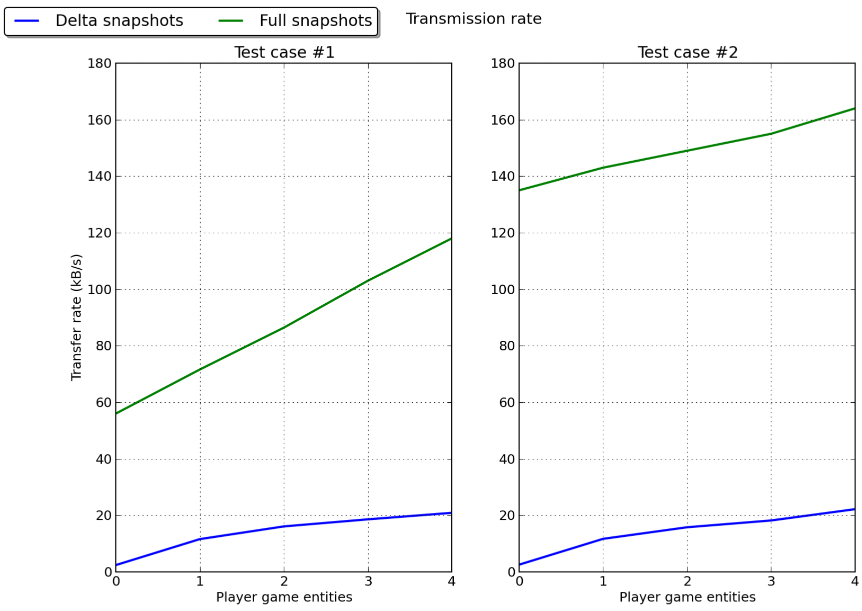


Figure 6.4: These figures display the difference in transmission rate for both test cases. The blue plot denotes delta snapshots and the green plot denotes full snapshots.

client. This was considered sufficient since the same amount of data would be transmitted to all connected clients. It is assumed that passing the full game state every tick will require considerable more network bandwidth than delta snapshots. On the other hand it could be more efficient for the master to serialize all its data directly into a String instead of enumerating all entities to check what entity has to be transmitted.

Figure 6.4 illustrates the effects by issuing delta snapshots instead of the full game state every tick. In test case #1 the propagation of the game world started at roughly 2.5 kB/s. This rate was mostly caused by the initial propagation of the full game world which has to be sent to all new clients. Since the master runs the simulation at 33 ticks per second the size of a delta snapshot totals to $\frac{1}{33} * 2.5kB/s = 0.0758kB = 75.8bytes$. The amount of transmitted data increases when additional dynamic game objects are added to the game since those are moving objects and thus need to be transmitted frequently in order to sync all the clients. Notice that this rate is not linear since at one or more points in time a game entity has been synced and don't need to be transmitted for one or several consecutive ticks.

The rate of growth for the full snapshots on the other hand is linear. This is due to the fact that each additional game entity is comprised of the same set of components. When these components are serialized into a snapshot the size of these snapshots will be equal. The size of a full snapshot equals: $\frac{1}{33} * 56kB/s = 1.697kB = 1697bytes$. This is about 22 times the amount of data passed when distributing delta snapshots.

Similar observations was made for test case #2. A game level four times larger (in terms of static game objects) than the previous case was distributed. The general trend for both delta snapshot graphs is very similar regardless of the amount of objects distributed. This was not the case for full snapshots. Actually the client in the Same network started to time out when distributing the full game world at 33 ticks per second. The only measured data point was at about 200 kB/s before the clients stopped to respond. The tick-rate in the master was adjusted down to 20 ticks per second in order to get measurable data. This is why the slope of the plot is steeper in the first graph.

As these graphs display passing the full game state at each tick does not scale very well. When the game world is increased by four dynamic entities the rate at which data is transmitted grows by 30 kB/s, and this is in addition to the static entities already being transmitted. The rate of data when passing delta snapshots grows by 20 kB/s for dynamic game entities. This means that the prototype would scale with static objects very well, since they are transmitted and synchronized only a few times. Transmitting a lot of dynamic game objects would eventually suffer the same fate as static game objects, but at a slower rate. Since each component decides whether it should be transmitted or not, a dynamic game object would only transmit its physical component, not its graphical representation or label component (as would be the case for full snapshots).

Since the delta snapshot model is based on principles introduced by Valve, it is interesting to compare these findings with an existing analysis on the Counter-Strike network model[2]. This paper shows that in a three person game on a Counter-Strike server the server approximately transmits data with an average rate of 6.470 kB/s. Similarly the prototype would transmit data with a rate of $3 * 2.5kB/s = 7.5kB/s$, to three clients. This can be assumed since this model transmits the exact same amount of data to all connected clients. While this certainly is an interesting comparison it is not a very fair comparison. In a typical Counter-Strike match there are more player specific state shared in additional to game state. Obvious state that is shared which exceeds the game state in the prototype is:

- In what direction a player is faced. This is most likely represented by a normalized vector relatively to the player position.
- If a player is firing his weapon
- When a grenade or flashbang is thrown, this entity has to transmit its position and velocity as well

What kind of skin a player has selected when joining the server is also reflected. However this similar to the random colour assigned to a player marble once a new player joins an existing game. This information is only transmitted once.

6.4 Prototype Profiling

Bottlenecks and poor performance could very well be caused by a non-optimal implementation. A lower frame rate for instance could be caused by an overly complex implementation of graphical objects. This could give a false positive when measuring the performance of Same by using the prototype.

To get a realistic view of how much CPU time is spent on overhead calculations in the prototype itself compared to Same the application was profiled using the profiling tools bundled with the Android (Android) Software Development Kit (SDK)[3]

6.4.1 CPU Time

CPU time spent was noticed on the tick rate, and the fps on the device. This was more apparent on Nexus S than the Nexus 4. By comparing their hardware, it was first assumed that this was caused by the slower GPU in the Nexus S. After removing code that required draw calls however, the FPS was still an issue (which also led to a lower tick rate). The issue was further investigated by using the "method profiling" tools in the SDK.

Dalvik Debug Monitor Server (DDMS)[4] is a tool included in the Android SDK. This tool have several metrics to display how much CPU time is spent in a given method. The profiling results in an *inclusive* and *exclusive* time. Exclusive time is the time spent in the method and inclusive time is the time spent in the method plus the time spent in any called functions. DDMS also measures the number of calls and recursive calls issued on a method.

The results from profiling the application is attached in Appendix C. As seen from this figure the application spends a fair amount of time in the *RequestHandler.run()*-method in Same. This was further traced down to the *UpdateStateRequest()*-method in the Master class. This is the method that distributes state updates to all clients. Even further investigation lead to the *State.getList()*-method which parses a String of all the connected clients into a Java Array. The String got parsed every time the master would propagate the state. An optimization could be to only parse and update this list given that there has been any changes to the number of clients added. As a test this parse call a got removed and a noticeable boost in performance was noticed on the Nexus One. As expected this broke the state propagation.

[3]Android DDMS - `http://developer.android.com/tools/debugging/ddms.html`
[4]DDMS - `http://developer.android.com/tools/debugging/ddms.html`

6.4.2 Dalvik Garbage Collector

The Dalvik Garbage Collector[5] uses a "Mark and Sweep" approach for freeing up memory[3]. One of the disadvantages by this approach is that normal program execution is suspended when the garbage collection algorithm runs. In later versions of Android (Gingerbread and beyond) the GC has been optimized to run on multiple cores, but for single core devices running real-time applications the periodically garbage collecting pause can be noticeable.

During the test executions it became apparent that the Garbage Collector periodically runs quite aggressively. The main reason for this would be object allocations on the game loop. DDMS has a tool for checking object allocations at run time. After running this tool for a while and polling active allocations several byte arrays of length 4112 allocated by Protocol Buffers got pointed out as aggressive memory allocators. This was further investigated by looking for similar cases on the Internet. One case suggested the allocation could be caused by the decoding and encoding of String objects. It is assumable that these allocations are necessary. After all the Protocol Buffers encodes and decodes a different String representation every tick, and avoiding garbage collection completely is hard if not impossible when working serializing and de-serializing complex data structures.

One method to generally mitigate issues with garbage collection would be to implement object pooling as described in Section 4.3. However this would be most beneficial for application specific object allocations and not something that should be implemented in Protocol Buffers itself.

6.5 Potential Bugs

This section contains a small set of various bugs encountered when implementing Same into the prototype. Keep in mind that these bugs could might as well be caused by a faulty or ineffective adaption of Same and not necessarily by the platform itself.

One major bug encountered was that clients did not manage to reconnect to the Same network after being disconnected. If a master failed during run time the Paxos routine would run, a client would be selected as the new master and simulation would continue with the remaining clients. However the recently dropped master would not be able to reconnect as a client. It is assumed that this was caused by an internal state not being cleaned after the initial master got dropped.

Some issues were also encountered when testing the prototype using the Eduroam network on campus. Occasionally when a master and a client got assigned an IP address on the same subnet (78.91.48.0-255 for instance) they refused to connect to each other. This issue was not encountered when the master and client were

[5]Dalvik - `http://en.wikipedia.org/wiki/Dalvik_%28software%29`

assigned addresses in different subnets. The weird thing is that this issue was not experience when using a private local network for testing. This leads to the conclusion that there might be an error specifically with the Eduroam network or similar open networks.

Neither of these potential bugs did affect the implementation or test phase. However they should be further investigated and if possible improved if a game is to use Same as its network platform.

6.6 Same Adaption

An aspect worthy to consider is how accessible Same was to be adapted into an application from a developer perspective. Several examples were provided with the library. These examples thoroughly explained usage of the Variables and how to set up a basic distribution of state. The examples even included a simple example of real-time state propagation. Implementing changes in the library itself on the other hand was not as non-trivial. Only small portions of the code was documented, though a lot of the code was self documenting by means of variable and method names. Since implementing changes in the library never was a large part of the problem statement in this thesis (neither was modifiability of the framework in the original thesis on Same) this insignificant issue wont be emphasized. The fact that Same is distributed as a Open Source library greatly simplified the process of acquiring the code and plugging it into the application.

6.7 Development and Test Environment

This section will give a summary of development and profiling tools used throughout the development of the prototype.

6.7.1 Software

The Eclipse Integrated Development Environment was the tool used for writing code and running tests. The main reason for using Eclipse is the support for and easy setup with the Android SDK. The Android SDK is the suite of tools which enables developing, deployment and debugging for Android devices.

Git[6] was used to version control the source files of this project. Git made it easy to create milestone versions of the software, and to roll back to previous versions if some undesirable feature was discovered. The dependencies this project relied on

[6]Git - `http://en.wikipedia.org/wiki/Git_%28software%29`

(such as Same and LibGDX) was shared on Github[7], which is a web front-end for Git repositories. Git was used to clone these repositories locally.

The Same code base was also shared as a Maven artifact. During the initial setup of the prototype project and its dependencies several benefits were discovered by using a Maven as the dependency management system. The process of deploying to Android devices got highly automated and dependencies kept themselves up to date. These benefits resulted in the prototype project being mavenized as well.

6.7.2 Testing

Sampling test data was achieved by creating a Java class purely for gathering statistics. This class comprised of several counters for each metric to be measured. Tick-rate was measured by attaching a world-update listener in the master. When the master updated its variable this listener got notified and incremented the tick-rate counter. The tick-rate was also validated at the clients, which similarly updated their perspective of the tick-rate when an update was issued by the master. Frame rate was measured at the master by incrementing a variable at each render call. After the sampling interval finished the measured data was smoothed over accumulated time. To get a realistic picture of how well Same performed, a virtual warm-up interval was set to 30 seconds. After the warm-up interval the statistics class would sample data for 60 seconds. This was assumed to be enough time to get a realistic view of the performance. The warm-up period was added to mitigate noise in the sampling process (caused by initializing classes and building the level). Garbage collection and method profiling was monitored by using the internal tools in the Dalvik Debug Monitor Service. These are tools bundled with the Android SDK.

Testing was carried out in a study room at school campus. Distance to the wireless access point which all devices was connected to was about 10 metres. This should be more than sufficient in order to assume that testing conditions were normal.

6.8 Evaluation

The game was created based on the architectural requirements specified in Section 4.2. This section aims to evaluate the game in regards of these requirements.

AR #1 Localized simulation
Every client is able to run its own simulation of the game world. If an update is delayed from the master the client may extrapolate the game world and simulate the expected positioning of game entities. This adheres nicely to the architectural requirement.

[7]GitHub - `http://github.com`

AR #2 Authoritative master

The master is in charge of facilitating all game state propagation. Clients attach listeners to the variable which contains the state update and the master is the only device writing to this variable. This implementation adheres to the concept of an authoritative server.

However there is no mechanism preventing a mischievous client to also write to the world state variable. This is not directly tied to this architectural requirement though but should be considered as a security flaw.

AR #3 Transient fail over

When a master failure is detected the simulation is temporarily suspended at all the clients. After a client is chosen as the new master all the other clients resume their simulation. The failure itself is not entirely transient since a small pause occurs. However after a few seconds the game can continue being simulated by a new master.

AR #4 Minimal network traffic

During the implementation phase there was a focus on minimizing network usage. Passing delta updates was a result of this design. However there are still other techniques that can be applied to further minimize network usage. Some of these techniques are discussed in Section 7.4.

AR #5 Gameplay based on physics

A physics engine helped implement this requirement and that proved very successful. Under normal operation and propagation the physical simulation was reflected at all the clients.

The prototype has fulfilled most of its original requirements and should be considered a valid candidate for a real-time multiplayer game.

Chapter 7

Conclusion and Further Work

Initially this project started out with the goal of evaluating and testing a framework for sharing state between Android devices. A simple physics based game was developed on top of this framework to test its capabilities and to reveal potential problems.

Throughout development it became apparent that the size of the state that got distributed and at what rate it got distributed was important factors that had an impact on the performance as whole in the Same network. As a result of this different techniques for achieving smooth gameplay at every client were implemented and tested.

This chapter summarizes the findings and results of this project, suggestions on features that could be implemented in Same and potential further work.

7.1 Conclusion

The prototype was designed to be a real-time game which had several concurrent moving entities that had to be reflected at all clients. Part of this project was to evaluate Same and to verify that its model is useful as a multiplayer platform.

Same proved to be a decent candidate for this type of multiplayer games except for a few quirks. One of the main features in Same is that the state is guaranteed to be propagated to all connected clients. For real-time physics based games this is not strictly necessary. The client can predict how entities should be simulated and fix their positions when a snapshot arrives. In fast paced multiplayer games it is common to distribute such "corrections" over UDP. The advantage of a higher rate of packets exceeds the disadvantage of some packets getting lost or received out of order.

For a turn based game Same would be a very good candidate. In this type of games the state is only distributed when changes occur, and not at the same rapid interval as in real-time games. For this case potential performance issues would be negligible.

7.2 Improvements to Same

As discussed in Section 5.4 several additions were made to Same in order to fully be able to adapt the framework as the network architecture for the prototype. The most important addition was the implementation of client differentiation. Since the network model in the game was designed as client-server (where the master was the equivalent to a server) this was necessary in order to handle input from the clients properly.

A feature that would be useful, which also would make the framework more appealing for real-time games would be a 'best effort' variable that could be implemented on top of UDP instead. This variable would propagate a state update to the master, and the master would blindly propagate the update to all its clients without listening for a callback. This kind of variable would be very useful for propagating delta snapshots and other kinds of intermediate data, and would not be as resource demanding as the regular variables.

As seen in Chapter 6 the rate of state propagation is highly dependent on the performance of the master. This was also emphasized in the original master thesis on Same. One solution to mitigate this problem would be to select the best performing master. This could be achieved by letting clients pass a metric upon state updates. This metric could be the average number of frames per second smoothed over some time or information from the operating system (a greater version number often entails a more powerful device). And then let Paxos choose a new master weighted on that metric.

7.3 Resulting Artifacts

A video of the game was created and published at `http://www.youtube.com/watch?v=rfICJUr0JNw`. This small video displays three clients connecting to the initial master. It shows how a game level is distributed by the master and generated on the fly at each client. A master disconnect is also demonstrated which shows the real power of the Same platform.

An open source library for handling entities and components was created throughout this project. It is hosted on Github: `https://github.com/aspic/libgdx-utils/tree/master/src/no/mehl/component`. The main advantage in this library as opposed to other entity libraries is the focus on distribution of entities. This includes the population of changed entities on component level. The library is currently in a very early stage and is mainly focused towards the implementation of the prototype.

7.4 Further work

The goal of this project was to create a real-time multiplayer game which adapted Same as its network model. Results show that Same is a decent candidate for such applications, but the prototype could be further improved.

As shown in Chapter 6 the size of the data distributed through the variable has a noticeable impact on performance in the system. One measure to minimize transmitted data would be to compress the JSON before it gets encoded by Protocol Buffers. Compression was not implemented due to the fact that most values passed through the variables are delta snapshots. These snapshots are mainly comprised of integers and floats, and would not benefit greatly by regular string compression.

Another measure to minimize size of transmitted data would be to lower the precision for these values. Positions and velocities in the simulation are represented by floating point numbers. These numbers are serialized and transmitted in their entirety. The numbers could be capped at a fixed set of decimal points to reduce the size of total transmitted data. This would be at the expense of data quality. If a client were to become the new master, its base data would be of lower granularity than the original simulation.

Bibliography

[1] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond. *Presented at GDC2001*, 2:30p, 2001.

[2] Mark Claypool, David LaPoint, and Josh Winslow. Network analysis of counter-strike and starcraft. In *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, pages 261–268. IEEE, 2003.

[3] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[4] Peter Eles. Capturing architectural requirements. 2005.

[5] Martin Fowler. *UML distilled*. Addison-Wesley Professional, 2004.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Abstraction and reuse of object-oriented design*. Springer, 2001.

[7] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[8] Björn Nilson and Martin Söderberg. Game engine architecture. 2007.

[9] Reusable Object-Oriented. Model-view-controller. 2003.

[10] Kjetil Ørbekk. Distributed shared objects for mobile multiplayer games and applications, 2012.

[11] Valve. Source multiplayer networking, 2012. [Online; accessed 03-March-2013].

[12] Peter Vorderer, Tilo Hartmann, and Christoph Klimmt. Explaining the enjoyment of playing video games: the role of competition. In *Proceedings of the second international conference on Entertainment computing*, ICEC ’03, pages 1–9, Pittsburgh, PA, USA, 2003. Carnegie Mellon University.

[13] Wikipedia. Lockstep, 2013. [Online; accessed 29-May-2013].

[14] Wikipedia. Real-time games — Wikipedia, the free encyclopedia, 2013. [Online; accessed 10-April-2013].

Appendices

Appendix A

Test Devices

This appendix lists the hardware and software specifications of the devices used in througout the implementation and testing phase.

Device	Nexus One
Display	800x480 px
Memory	512 MB
CPU	1 GHz Qualcomm Scorpion
GPU	Qualcomm Adreno 200
OS	Android 2.3.6

Table A.1: Nexus One

Device	Nexus S
Display	800×480 px
Memory	512 MB
CPU	1 GHz single-core ARM Cortex-A8
GPU	200 MHz PowerVR SGX 540 GPU
OS	Android 4.1.2 (CyanogenMod)

Table A.2: Nexus S

Device	Nexus 4
Display	1280×768 px
Memory	2 GB
CPU	1.5 GHz quad-core Krait
GPU	Adreno 320
OS	Android 4.2.2

Table A.3: Nexus 4

Device	Asus UX32VD
Display	1980x1200 px
Memory	4 GB
CPU	Intel Core i7-3517U 1,9 GHz
GPU	Intel HD 4000
OS	Ubuntu 13.04

Table A.4: Asus UX32VD

Appendix B

Tests Results

This appendix contains test data sampled when benchmarking the application.

Master device	Clients	avg. FPS	avg. tick-rate	Entities
Nexus One	1	24.51	23.43	6
	2	20.73	19.33	7
	3	17.90	17.11	8
	4	14.40	13.88	9
Nexus S	1	38.45	38.38	6
	2	29.43	29.26	7
	3	24.40	24.30	8
	4	20.11	20.01	9
Nexus 4	1	54.35	43.26	6
	2	50.08	40.08	7
	3	45.98	37.05	8
	4	42.38	32.73	9
Asus Laptop	1	60.00	60.00	6
	2	60.00	59.96	7
	3	59.96	59.88	8
	4	59.45	58.81	9

Table B.1: Master performance measured in-game

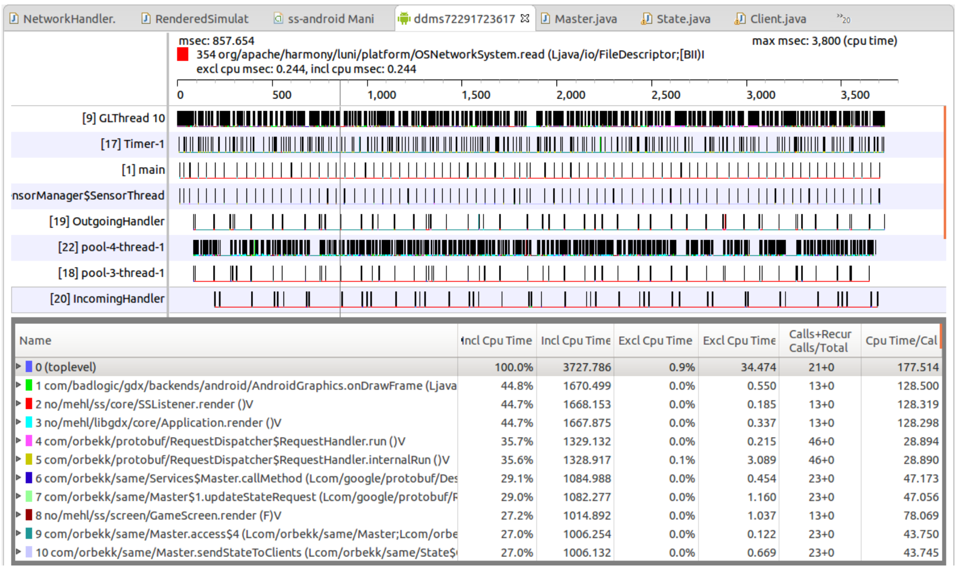
Master device	Clients	avg. FPS	avg. tick-rate	Entities
Nexus One	1	20.55	20.43	21
	2	17.56	16.54	22
	3	15.91	13.26	23
	4	14.40	13.88	24
Nexus S	1	28.03	26.95	21
	2	23.23	22.80	22
	3	20.65	18.72	23
	4	17.51	16.24	24
Nexus 4	1	50.2	42.26	21
	2	47.69	39.57	22
	3	44.12	38.29	23
	4	42.38	37.12	24
Asus Laptop	1	60.00	60.00	21
	2	60.00	60.00	22
	3	59.50	59.12	23
	4	59.52	59.10	24

Table B.2: Master performance measured in-game

Appendix C

Profiling Results

This appendix contains the profiling results from the Dalvik Debug Monitor Server.



msec: 857.654 max msec: 3,800 (cpu time)

🟥 354 org/apache/harmony/luni/platform/OSNetworkSystem.read (Ljava/io/FileDescriptor;[BII)I
excl cpu msec: 0.244, incl cpu msec: 0.244

Name	Incl Cpu Time	Incl Cpu Time	Excl Cpu Time	Excl Cpu Time	Calls+Recur Calls/Total	Cpu Time/Cal
0 (toplevel)	100.0%	3727.786	0.9%	34.474	21+0	177.514
1 com/badlogic/gdx/backends/android/AndroidGraphics.onDrawFrame (Ljava	44.8%	1670.499	0.0%	0.550	13+0	128.500
2 no/mehl/ss/core/SSListener.render ()V	44.7%	1668.153	0.0%	0.185	13+0	128.319
3 no/mehl/libgdx/core/Application.render ()V	44.7%	1667.875	0.0%	0.337	13+0	128.298
4 com/orbekk/protobuf/RequestDispatcher$RequestHandler.run ()V	35.7%	1329.132	0.0%	0.215	46+0	28.894
5 com/orbekk/protobuf/RequestDispatcher$RequestHandler.internalRun ()V	35.6%	1328.917	0.1%	3.089	46+0	28.890
6 com/orbekk/same/Services$Master.callMethod (Lcom/google/protobuf/Des	29.1%	1084.988	0.0%	0.454	23+0	47.173
7 com/orbekk/same/Master$1.updateStateRequest (Lcom/google/protobuf/R	29.0%	1082.277	0.0%	1.160	23+0	47.056
8 no/mehl/ss/screen/GameScreen.render (F)V	27.2%	1014.892	0.0%	1.037	13+0	78.069
9 com/orbekk/same/Master.access$4 (Lcom/orbekk/same/Master;Lcom/orbe	27.0%	1006.254	0.0%	0.122	23+0	43.750
10 com/orbekk/same/Master.sendStateToClients (Lcom/orbekk/same/State$	27.0%	1006.132	0.0%	0.669	23+0	43.745

Figure C.1: Profiling results.