Distributed Shared Objects for Mobile Multiplayer Games and Applications

Kjetil Ørbekk

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Svein Erik Bratsberg, IDI

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

Abstract

Mobile gaming for smartphones has gained huge popularity in recent years and has become a big industry. In 2011 Google Play passed 10 billion downloads. The game Angry Birds alone has been installed on devices more than 50 million times.

Increased performance of mobile devices means that new development techniques can be used to make more advanced games. This project presents a new technique to develop multiplayer games using a distributed system.

Multiplayer games often use a client-server architecture in order to communicate. Many games present a *shared game universe* to the players. Updates to the game state must be performed *consistently*, otherwise discrepancies in the game state may appear to the players.

Mobile devices are unreliable servers for several reasons. First, they use a Wi-Fi connection that may not be reliable. Second, the devices have limited amounts of resources. The operating system may need to terminate the server, e.g., to recover memory for important services such as receiving phone calls. Lastly, if the person with the server chooses to stop playing, the game cannot proceed.

A distributed system can prevent unreliable devices from causing failures. When a device becomes unavailable, all its responsibilities should be transferred to other devices in the system.

This project presents the distributed system *Same* for sharing state in mobile multiplayer games. *Same* is a peer-to-peer system with a master device that automatically recovers from failure. When a master fails, the system may enter an inconsistent state. Recovery is performed by electing a new master that restores consistency by comparing the state of all the clients in the system. The system is presented as a general model for sharing state and implemented for the Android operating system.

Same is evaluated using benchmarks and an example application. Updates are performed consistently and distributed to all the connected devices. *Same* is shown to be a viable alternative to a standard client-server architecture.

Sammendrag

Mobilspill for smarttelefoner har blitt veldig populært de siste årene, og dette har blitt en stor industri. I 2011 passerte Google Play 10 milliarder nedlastinger. Spillet Angry Birds har alene mer enn 50 millioner nedlastinger.

Økt ytelse i mobile enheter gjør at nye teknikker kan benyttes for å lage mer avanserte spill. Dette prosjektet presenterer en ny teknikk for å utvikle spill for flere spillere ved å bruke et distribuert system.

Spill for flere spillere bruker ofte en klient-tjener-arkitektur for å kommunisere. Mange spill presenterer et *delt spillunivers* for spillerene. Oppdateringer til spillets tilstand må gjennomføres på en *konsistent* måte, ellers kan det oppstå avvik i tilstanden til de forskjellige spillerene.

Mobile enheter er upålitelige tjenere av flere grunner. For det første bruker de en trådløs wi-fi-tilkobling som kan være upålitelig. For det andre har enhetene begrensede ressurser. Det kan være nødvendig for operativsystemet å avslutte tjeneren, f.eks. for å frigjøre ressurser til viktige tjenester som å motta samtaler. For det tredje så kan personen med serveren velge å slutte å spille, og da kan ikke spillet fortsette for resten av spillerene heller.

Et distribuert system kan hindre feil når man har upålitelige enheter. Når en enhet blir utilgjengelig må alt ansvaret til enheten flyttes til en av de andre enhetene i systemet.

Dette prosjektet presenterer det distribuerte systemet *Same* for å dele tilstand i mobile spill for flere spillere. Systemet er basert på en peer-til-peer-arkitektur med en master som automatisk gjenoppretter normal tilstand når enheter feiler. Når en enhet feiler kan det oppstå inkonsistens i systemet. Gjenoppretting skjer ved at en ny master velges. Den nye masteren gjenoppretter tilstanden ved å kombinere tilstanden til alle klientene i systemet. Systemet blir presentert som en generell modell for å dele tilstand og som en implementasjon for operativsystemet Android.

Same blir evaluert med ytelsesmålinger og et eksempelprogram. Oppdateringer blir gjort på en konsistent måte og distribuert til alle enheter tilkoblet systemet. Det blir vist at *Same* er et mulig alternativ til en standard klient-tjener-løsning.

Acknowledgments

I would like to express my appreciation for my supervisor, Professor Svein Erik Bratsberg. He has given me a lot of flexibility, as well as great advice when I was stuck or about to make a bad decision. Thanks for giving me the opportunity to shape my own project.

I would also like to thank Erlend H. Hamberg, Jessica Shindo and Katharina Kufieta for giving me helpful comments on this report, and Trygve André Tønnesland and Simen Andresen for providing test devices. A special thanks to Henning Smistad for helping me with illustrations and presentation.

Contents

1 Introduction and Motivation **1**
- 1.1 Problem Description . . . 1
- 1.2 Project Goal . . . 2
- 1.3 Document Structure . . . 2
- 1.4 Requirements . . . 2

2 Distributed Systems Background **5**
- 2.1 Properties of Distributed Systems . . . 5
- 2.2 Master Election . . . 6
- 2.3 Peer Discovery . . . 10
- 2.4 Multicast . . . 11
- 2.5 Related Distributed Systems . . . 12

3 Architecture and Design **18**
- 3.1 Programming Model . . . 18
- 3.2 State Model . . . 20
- 3.3 System Overview . . . 21
- 3.4 System State . . . 23
- 3.5 Participant Overview . . . 24
- 3.6 Master Takeover . . . 28

4 Implementation **32**
- 4.1 Concurrency Model . . . 32
- 4.2 Thread Pool Starvation . . . 34
- 4.3 Remote Procedure Calls . . . 34
- 4.4 Toward a Faster RPC for Mobile Devices . . . 35
- 4.5 RPC Performance . . . 38
- 4.6 Paxos Implementation . . . 39
- 4.7 Dependency Injection . . . 41

5 Evaluation and Experiences **44**
5.1 Example Application 44
5.2 Benchmarks 45
5.3 Development Environment 48
5.4 Evaluation 49

6 Conclusion and Further Work **51**
6.1 Resulting Artifacts 51
6.2 Further Work 52
6.3 Lessons Learned 52
6.4 Conclusion 53

A Class Diagram **56**

B Test Devices Overview **58**

List of Figures

2.1 The Bully Algorithm 7
2.2 Paxos 9
2.3 Operating System Distributed Shared Memory 13
2.4 Middleware Distributed Shared Memory 13

3.1 System Overview 22
3.2 Master takeover 29

4.1 Protobuf RPC server 37

5.1 Example Application 45
5.2 Update latencies 46
5.3 Master Takeover Time 47

Chapter 1

Introduction and Motivation

Mobile gaming for smartphones has gained huge popularity in recent years and has become a big industry. In 2011 Google Play passed 10 billion downloads[1]. The game Angry Birds has been installed on devices more than 50 million times[2].

As the performance of mobile devices increase, new development techniques can be used to make more advanced games. Advances in graphics technologies has made it possible to create 3D games on mobile devices. Increasing amounts of RAM lets us run larger programs with more graphics content. Widespread mobile internet enables multiplayer games and social features, such as instantly comparing your high score with friends online.

1.1 Problem Description

Multiplayer games often use a client-server architecture in order to communicate. Many games present a *shared game universe* to the players. Updates to the game state must be performed *consistently*, otherwise discrepancies in the game state may appear to the players.

A common solution to this challenge is to use a client-server architecture where one of the devices acts as the server. However, mobile devices are unreliable servers for several reasons. First, they use a Wi-Fi connection that may not be reliable. Second, the devices have limited amounts of resources. The operating system may need to terminate the server, e.g., to recover memory for important services such as receiving phone calls. Lastly, the person with the

[1] `http://googleblog.blogspot.com/2011/12/10-billion-android-market-downloads-and.html`

[2] `https://play.google.com/store/apps/details?id=com.rovio.angrybirds`

server may choose to stop playing. If the server becomes unavailable, the game will fail.

A distributed system can prevent unreliable devices from causing failures. The responsibility of any device in the distributed system must be transferred to other devices when a device fails. However, implementing a distributed system is a challenging task because one must take into account different causes of failure in a concurrent environment.

1.2 Project Goal

The goal of this project is to show that distributed objects can be used to create reliable real-time multiplayer games for mobile devices. In order to do this, a distributed shared object model for mobile devices is proposed. The model is based on a peer-to-peer architecture where one of the devices acts as a master. When the master fails, another client takes over and assumes the master responsibility. The object model consists of a set of Javascript Object Notation (JSON) values which are synchronized consistently to all the devices of the network. The suggested object model has been implemented on Android and is called *Same*.

With a system that can be used to synchronize state with different types of games, game developers can enjoy the advantages of distributed systems without needing to develop a specialized system for their games.

1.3 Document Structure

The rest of this chapter describes some requirements of our system. Chapter 2 gives background information on distributed systems and describes related systems. Chapter 3 describes the architecture of *Same* and Chapter 4 contains implementation details and challenges that were encountered. Chapter 5 presents our experiences when using *Same* and performance analysis. Conclusions and further work are presented in Chapter 6.

1.4 Requirements

Developing multiplayer games requires network programming and synchronizing state across all devices running the game. Accounting for unreliable devices is a challenge, because there can be no single trusted server that handles synchronization.

Can a framework synchronize state between devices automatically? The following properties should be satisfied by such a system:

Localized Architecture The framework will be used in a single geographical location by multiple participants. Many existing games use the internet to communicate via a reliable central server. Instead, this projects targets devices that are connected to the same local Wi-Fi network.

Low Latency The system should be responsive enough to support synchronization of state in real-time multiplayer games. Our goal is a throughput of at least 25 updates per seconds. This is equivalent to a 40ms update latency on average.

Scaling The system should scale to a modest number of mobile devices. Because of the localized architecture, supporting 10 devices will be considered sufficient.

Accounting for Unreliable Devices If a device disconnects or otherwise becomes unavailable to the remaining devices in the system, the system should continue to function, possibly removing the lost participant from the system.

Programming Model The system should offer a programming model that allows development of several types of games.

Discovery and Connection There should be an easy way for new devices to connect to the system without any configuration of the devices or network equipment.

1.4.1 Trust

Another important feature of the system is some type of security model: Which devices should be allowed to join? Who is allowed to read the state or perform updates?

Without built-in security features, cheating can be done by accessing data that should not be available to the player. For instance, in a card game any one player should not see the cards of other players. If the game synchronizes all cards to all participants, a player may gain access to another player's cards by modifying their client.

Unfortunately, the scope of this project is limited. Therefore it is assumed that all the clients in the system are trustworthy. The networks created by *Same*

are completely open for anybody to join and to inspect all values in the system, such as the card objects in a card game. Security features are left as future work.

Chapter 2

Distributed Systems Background

This chapter presents background material for distributed systems and related systems.

2.1 Properties of Distributed Systems

Although distributed systems are used for many different purposes, e.g., databases; file systems and distributed computations, many aspects of their architecture and programming model can often be described using some of the following properties.

Atomic Transactions Whether a sequence of operations can be performed *atomically* by the system, i.e., either none or all the operations are performed, and no intermediate state may be observed.

Sequential Consistency Whether two operations are guaranteed to be performed in the same order on all nodes. If two writes, w_1 and w_2 are performed, these writes must be performed in the same order on all nodes in a system with sequential consistency.

Fault Tolerant Whether mechanisms in the system help recover from failure. In distributed systems this is implemented with redundant nodes such that a failing node can be replaced by another node.

Peer-to-Peer Systems without a centralized server are known as peer-to-peer systems. Any computer in such a system is known as a peer, and all peers have similar roles.

These properties are not well defined, because they have different meanings in different systems. When analyzing distributed systems, one must consider what terms such as "consistency" means in the context of the system at hand.

The notion of fault tolerance is particularly difficult. Many computer systems have some level of error handling, but none of them are completely resistant to all types of errors. For instance, the Paxos algorithm handles approximately $n/2 - 1$ simultaneous failures, where n is the number of devices in the system. A storage system that uses parity bits in order to provide redundancy handles only one failure. Both of these system are fault tolerant, to different extents.

An alternative measure of fault tolerance is availability. Instead of measuring how many errors the system can handle before failing (possibly irrecoverably), a system may simply become unavailable when it cannot handle more errors. Availability is often characterized by "number of 9s", e.g. five 9s means that the system is available 99.999 percent of the time.

2.2 Master Election

Fault tolerant systems should not have single points of failure. Sometimes distributed systems have participants with special responsibilities. If such a component fails, its responsibilities must be transferred to another participant.

A participant with an important responsibility in the system is sometimes referred to as the master. The process of choosing which participant should become the master is called master election.

2.2.1 Bully Algorithm

The Bully Algorithm [1] is a master election algorithm based on an ordering of the process IDs in the system. When the master fails, a process is allowed to become the master if it has the highest ID. A master election is shown in Figure 2.1. The algorithm has the following steps:

1. If process p has the highest process ID, it becomes the master directly by sending coordinator messages to all processes. Otherwise p starts the election by sending elect messages to all processes $q_i > p$.

    (a) If process q_i responds with an answer message, p waits for the election to finish. If it takes too long, it will restart the election.

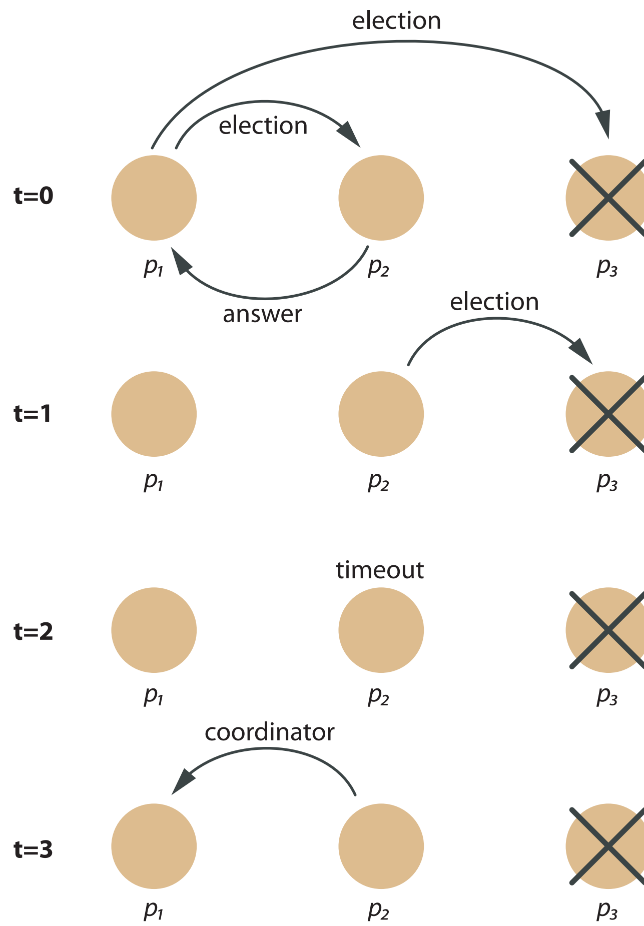


Figure 2.1 – A master electing with the Bully Algorithm. Process p_3 has failed, and p_2 should become the new master.

   (b) If all q_is fail to respond, p assumes that it has the highest ID and sends a coordinator message to become the master.

2. Any process that receives an elect message responds with an answer message and starts the process from step 1.

   A process that joins the network is allowed to start an election immediately. If it has the highest process ID it will become the coordinator. This is why it is called the Bully Algorithm.

   It is possible to come up with alternative values instead of process IDs, as long as the processes can be ordered. For instance, some processes may have more computing power than others. It may be desirable to have high-performing master nodes. By ordering the processes by computing power, the the highest performing node will become the master.

2.2.2 Paxos

Paxos [4] is a more general consensus algorithm, that allows a set of processes to agree on values. The Paxos algorithm is based on different roles within the system. A process may be a proposer, acceptor or a learner, or any combination of these. In this project, all processes implement all of the roles.

The process that wants to establish consensus on a value is called the *proposer*. The processes that receive a proposal are called *acceptors*. The *learners* receive values that have been accepted and stores them for use in the system. A successful iteration of Paxos is shown in Figure 2.2.

The Paxos algorithm is performed as follows. Let N denote the number of processes in the system.

1. Process p wishes to propose the value v. It sends a proposal to all the acceptors. The proposal contains an *id* k, which is the highest proposal number p has seen so far.

   (a) An acceptor A_i responds with a *promise* if k is bigger than the largest proposal A_i has previously seen, say k_i. Otherwise it responds with a rejection containing k_i.

   If $n > N/2$ of the acceptors responded with a promise, p proceeds.

   (b) Otherwise p retries from 1 with k set to the maximum $k_i + 1$.

2. After $n > N/2$ promises, p sends an accept request with the proposal number to all the acceptors.

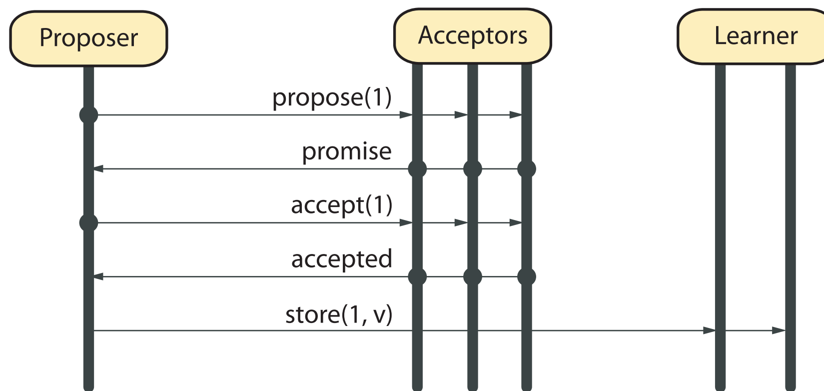


Figure 2.2 – A successful iteration of the Paxos algorithm with one proposer, three acceptors and two learners.

(a) An accept request is handled identically to a proposal request by the acceptors, except that it returns *acceptance* on success.

If $n > N/2$ of the acceptors responded with acceptance, p proceeds.

(b) Otherwise, p retries from 1 with k set to the maximum $k_i + 1$.

3. When a value has been accepted, consensus has been established and p may write the value to the learners.

Master Election with Paxos

In this project, Paxos has been used for master election in the following way: A participant p tries to establish a consensus with the Paxos algorithm with id k. If it succeeds, it may send a broadcast message to all the participants advertising that p is the new master with number k. If it fails, it tries again after a timeout.

When a process receives a message that advertises a new master, it aborts its master election process in order to avoid starting another master.

If unlucky timing occurs, the proposal number is used to identify the correct master. Because each proposal number is unique, one of the masters have the highest such number. The clients follow the master with the highest master number, which is always the most recently elected master.

2.3 Peer Discovery

In distributed systems systems one needs a mechanism to find other peers in the system. The procedure of finding other peers in the system to connect to is known as peer discovery [7].

In client-server architectures, clients may join the system by connecting directly to the server. It is not always as easy in peer-to-peer systems because they lack centralized servers. Additional challenges arise when attempting to handle different types of networks and failure situations. For instance, if only a single peer is known, it is not possible to join the network if that peer fails.

2.3.1 Static Discovery

A very simple form of peer discovery is to maintain a configuration file containing the peers of the system. When the system starts, it connects to the other peers found in its configuration file. This method is suitable for networks with a constant set of peers, where each peer has an address that never changes.

2.3.2 Directory Server

A *directory server* maintains a list of currently available peers. When a peer needs to join the network it contacts the directory server to get a list of peers to connect to. If the set of peers frequently change, a directory server can maintain a view of the network. This process is automatic and there is no need to change a configuration file when changes occur in the network.

2.3.3 Member Propagation

If a distributed network is very large, maintaining a complete list of all peers may not be feasible. Instead a partial list of member can be stored. When a peer needs to join the network, it discovers a subset of the peers of the network. If a complete peer list is required, the peer can recursively query other peers for their member lists in order to build a complete view.

2.3.4 UDP Broadcast

If the network is very small and located on a LAN, it is possible to use UDP broadcast messages for peer discovery. Peers that are connected to the network listen for broadcast messages on a specified port. When a new peer needs to join the network it sends a UDP broadcast message to the same port. This notifies

the peers in the network that a new peer is available. A message will be sent to the new peer containing information that lets the new peer join.

One may consider UDP as too unreliable to use for communication between nodes because it has very little failure handling. During discovery, however, no harm is done if a discovery packet goes missing: The client will simply retry.

2.4 Multicast

In computer networking, sending the same message to a group of receivers is known as multicast. Multicast can be implemented in various forms, e.g., in the networking stack, in the operating system, in the transport libraries or in software.

Only software multicast is described here, because reliable network-level multicast depends on support and configuration of networking equipment. Depending on special network setup is not acceptable because most users do not have the necessary equipment and skills.

2.4.1 Software Multicast

Reliable multicast can be performed in software with a TCP connection to each client. When the server wishes to send a message, the message is simply sent to each client separately.

With a low number of devices, software multicast is relatively fast.

With a high number of devices, software multicast will be slower, but may be performed using other algorithms.

2.4.2 Group Communication

Group communication systems provide reliable messaging in distributed systems [6]. Such systems utilize different techniques such as network or software multicast. When a message cannot be delivered, the group communication system will attempt to resend it.

A *group* is defined as a set of processes that send messages to each other. When one of the processes sends a message, it is delivered to all the other processes within the group.

Spread Toolkit

Spread Toolkit[1] is an open source group communication system. It supports several different schemes for message delivery, including multicast and group communication.

An application uses *Spread* by communicating with a local *Spread* server. This server must be started separately from the application. The *Spread* API interacts with the local server, which in turn sends and receives messages to other *Spread* servers.

This is similar to the Android architecture of the system proposed in this project: The system runs in a separate process and exposes an Android *service*. An Android application interacts with the system using message passing.

Again, because only a low number of devices will be used in this project, software multicast should be efficient enough. In a distributed system with many devices, *Spread* could be used to simplify the communication between processes.

2.5 Related Distributed Systems

Many distributed systems for state sharing exist. This is a summary of some related systems.

2.5.1 Distributed Shared Memory

A distributed shared memory (DSM) is a virtual memory that provides shared memory semantics to processes in a distributed system [5]. Each process has access to reading and writing to the shared memory. Distributed shared memory may be implemented in the operating system or as a middleware. A DSM implemented in the operating system level provides implicitly shared memory to processes, as shown in Figure 2.3. The middleware approach provides a programming interface to the DSM, in which a memory unit is shared explicitly by the programmer. This is shown in Figure 2.4.

Operating System Level DSM

Operating systems handle read and write requests to virtual memory addresses. Each available virtual address corresponds to a *word* on a page. A subset of the pages are mapped into the operating system as space allows.

[1]http://www.spread.org

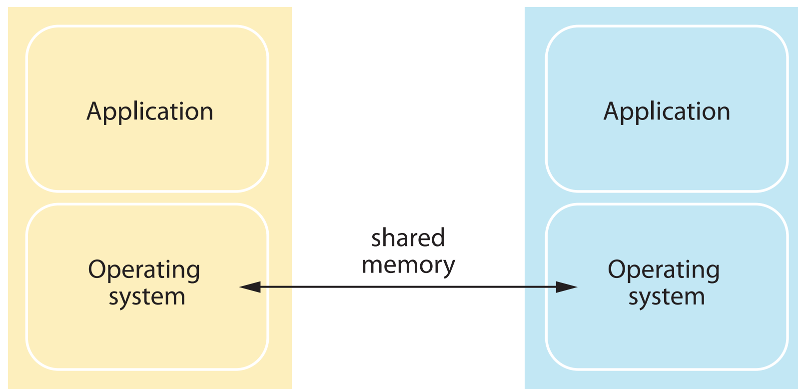


Figure 2.3 – Distributed Shared Memory implemented in the Operating System.

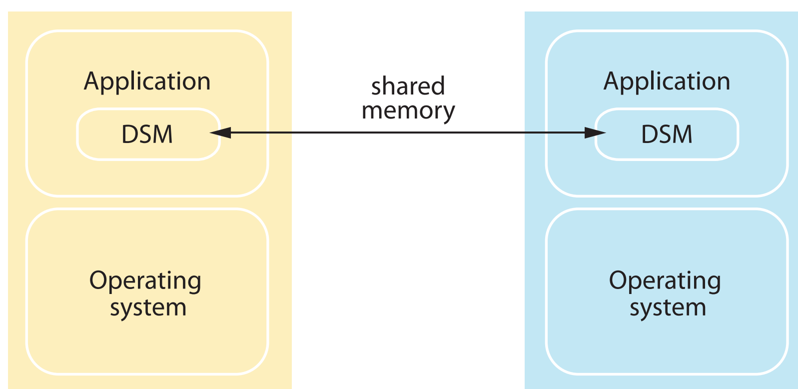


Figure 2.4 – Distributed Shared Memory implemented as a library used by the application.

In a DSM, a page can be written to by at most one process at a time, otherwise inconsistency will arise. When a process runs a read or write operation, the DSM looks up the address of the process holding the correct page. The page is unmapped from the process holding it, sent over the network to the caller process, mapped into the memory of the caller operating system and then the operation is performed.

A problem with operating system level DSM occurs when unrelated variables are stored on the same page. If two variables are stored on a single page and two processes write to the variables, the page must be sent over the network even if the writes are unrelated – a huge decrease in performance.

Middleware Based DSM

Middleware based DSMs provide a programming interface to the distributed memory. A programmer explicitly declares the variables they want to share.

Memory Model

Middleware based DSMs are not subject to the traditional memory interface provided by operating systems. Instead, middleware based DSMs provide a memory model to the application developer and may have different synchronization and consistency guarantees than operating systems do. For instance, a middleware based DSM may share logical objects rather than pages. This alleviates the problem of synchronizing unrelated variables: A shared object may contain unrelated variables, but it is not as likely.

Error Handling

Memory operations at the operating system level are assumed to always succeed. Therefore, operating system level DSMs have no flexibility when handling errors. If a program requests a memory location on a page that is temporarily unavailable, the operating system must either terminate the program or block until the node becomes available.

In contrast, middleware based DSMs may expose their error handling to the application developer. If the memory operation cannot be performed, the application is allowed to handle the error.

Latency

Memory operations in DSMs may require network operations, which are orders of magnitudes slower than local memory in operations. In cases of high latency,

an asynchronous interface may be preferable to traditional blocking memory operations.

2.5.2 Linda

Traditional distributed systems may use shared state (e.g. DSM), message passing or remote procedure calls to communicate between processes in a distributed system. *Linda* introduced a new paradigm that decouples processes, yet it supports coordination and shared storage [3].

The *Linda* coordination model is based on shared, distributed state, but differs from shared memory. In DSM the shared state is accessed directly: a shared variable can be read or written to. *Linda* offers special operations with semantics that are designed to make concurrent access easier to manage.

Shared state in *Linda* is represented as typed ordered tuples, and the shared state is called a *tuple space*. The tuples in a *tuple space* are immutable but they may be extracted and replaced with another, modified tuple.

Linda has the operations out(), in() and read(). For simplicity, untyped version of the operations is presented here.

- out($p_1, \ldots, p_i$) writes the tuple $(p_1, \ldots, p_i)$ to the *tuple space*. Duplicate values are allowed.
- in($p_1, \ldots, p_i$) extracts a tuple from the *tuple space*. In this case the parameters are used as a pattern to find matching tuples. For instance, the operation in(food, mouse, any) will extract a tuple such as (food, mouse, cheese) from the *tuple space*.
- read($p_1, \ldots, p_i$) is equivalent to in($p_1, \ldots, p_i$) except that the tuple is not extracted from the *tuple space*.

The *Linda* operations have several advantages over direct access to shared state. For instance the race condition in Listing 2.1 can be avoided in *Linda* as demonstrated in Listing 2.2. If in() is executed concurrently, one of these calls will atomically extract a tuple, and the other call is blocked until the a matching tuple is written back to the *tuple space* with out().

Compared to message passing and RPC, the advantage of *tuple spaces* is that processes are *loosely coupled*, and that *tuple spaces* offer persistent storage. For instance, an online bookstore could have two different processes: A web server that lets users order books and an order processing server that processes payments and schedules shipping. When a user orders a book, the web server puts a tuple in the *tuple space* representing the order. Whenever the order

Listing 2.1 – Race condition when incrementX() is called by concurrent processes.

```
void incrementX() {
    int x = readSharedVariable("x");
    writeSharedVariable("x", x + 1);
}
```

Listing 2.2 – IncrementX() using Linda: No race condition.

```
void incrementX() {
    (unused, x) = in("x", any);
    out("x", x + 1);
}
```

processing server is ready to process the order, it reads the tuple. When the order has been processed, it can remove the tuple. If the order processing server fails in this design, it can be restarted without losing any orders. (Additional failure recovery is necessary in order to avoid processing payments twice, otherwise customers may be unhappy with the system.)

2.5.3 JavaSpaces

JavaSpaces is a modern implementation of *Linda*'s *tuple space* [2]. Java objects are stored in *spaces* by the processes in the system. Similar to *Linda*, a *space* is a distributed repository of objects and processes use the *spaces* for persistent storage and for communication.

The basic operations on *spaces* are similar to the *Linda* operations: read, write and take. *JavaSpaces* adds many features to the *Linda* model, such as distributed transactions and the notify command.

JavaSpaces for Mobile Games

The distributed object model to be presented in Section 3 supports shared global state between devices. When an object is updated, it is immediately synchronized to all the other devices. These semantics are important in order to share global state in multiplayer games.

JavaSpaces' development model can be used to simulate the operations in our object model. The following is a naive translation of *Same* semantics to *JavaSpaces*. Refer to Section 3.1.1 for more information on these operations.

1. All variables that will be used are initially added to a *space* with a special *nil* value. This is done in order to differentiate between undefined and unavailable variables.

2. A write operation is performed by first taking the object from the *space* and then writing the new version. This ensures that variables will be updated sequentially: If two nodes try to perform a write concurrently, only one of the take operations will succeed. The other node has to wait until the first node writes back its new value.

3. An update operation is performed by reading the object from a *space*. If the object is not available, some other device has taken it and is writing a new version. When that device returns the object, the update will finish.

4. Update notification can be performed with the notify operation.

These operations work under normal conditions, but do not handle failures. If a node takes an object and then fails before writing it back to the *space*, the object will be permanently lost. In addition, no one may read an object while it is being updated.

We can attempt to solve these problems by introducing lock objects: When a write operation is performed, a special lock object is taken from the *space*. In case of failure, the lock must expire after a certain time. This way another node is allowed to perform its write operation after a certain timeout.

Alternatively, *JavaSpaces* supports transactions, which can be used as an alternative to the lock system mentioned above. Simply perform each take-then-write in a transaction to make the write operation safe.

The operations above (1-4) simulate *Same* semantics, but there are differences in how the operations are performed.

1. *JavaSpaces* is a distributed system designed to help developers build distributed applications that run on multiple servers [2]. *Same* is designed to run on mobile devices.

2. *JavaSpaces* Transactions are performed using *two-phase-commit*s. This is a much more expensive operation than a *Same* write, which is performed with a single query to the master.

3. All *JavaSpaces* operations may require accessing remote data, but *Same* automatically broadcasts the entire state. Therefore, accessing variables is less expensive in *Same*. *JavaSpaces* is excellent if each node only accesses a small subset of the available objects. *Same* is better for multiplayer games where each node requires all the shared objects.

Chapter 3

Architecture and Design

In this chapter the distributed object system named *Same* is proposed. First, the programming interface is presented. This is a brief introduction that covers only some of the features. Second, the internal state model and semantics are discussed. Third, the system architecture and design of the *Same* implementation are briefly covered. Lastly, failure handling and master recovery is presented in detail.

3.1 Programming Model

The *Same* programming model exposes a service that allows sharing state with several devices. State is shared within a network of connected devices. A discovery service lets users automatically discover available networks or register their own.

Same stores objects for the participating clients. An object is stored as a JSON value. The JSON values are serialized from Java objects using the Jackson[1] library.

A client stores an object by writing it to a *Same* variable. When an object is written, it is automatically synchronized to all the other clients in the network. Each object carries an internal *version* field, which is used to maintain sequential consistency on a per-object level. If a write is requested on an outdated object, it is rejected by the master.

In normal operation *Same* offers low-latency synchronization. When devices disconnect or otherwise fail, delays may occur until a new master is elected and consistency is verified. A signal is sent to the client application and update requests will block until the system has resumed normal operation.

[1] `http://jackson.codehaus.org/`

Listing 3.1 – Updating a variable with the Same programming interface.

```
void setVariableExample(VariableFactory vf) {
    // Create the String variable "hello" in Same.
    Variable<String> hello = vf.create("hello", String.class);

    // Assign the variable. Returns an object that represents an
    // asynchronuous operation (similar to a Future).
    DelayedOperation op = hello.set("Hello, Same!");

    // Wait for 'op' to finish.
    op.await();
    if (op.isOk()) {
        // Operation succeeded.
    }
}
```

3.1.1 The Variable Class

A programmer using *Same* accesses shared objects with the Variable class. A Variable object is created with a user-supplied name and type. When the user interacts with the variable, it communicates with the *Same* backend to provide synchronization. A Variable supports the following operations:

- The get() operation returns the current value of the variable. This value is initially set to the most recent value the local client has seen, but is never changed unless the user invokes the update() method.

- The update() method updates the object in the same manner as the initial update. Having an explicit update() method forces the user to acknowledge a new value before overwriting it. For instance, if the user calls set() twice, only the first invocation will succeed. The second invocation attempts to update an outdated variable, which will fail. The user has to update the variable between each call to set().

- The set() method tries to set the variable to a new object. This operation is asynchronous and returns a DelayedOperation. The DelayedOperation class is similar to a *Java* Future[2], except it does not throw as many exceptions. Listing 3.1 is an example of interaction with a String variable with the name "hello".

[2] java.util.concurrent.Future

Listing 3.2 – IncrementX() using Same: No race condition, because set() fails if x was updated concurrently.

```
void incrementX() {
    Variable x = variableFactory.create("x", Integer.class);
    boolean updated = false;
    while (!updated) {
        x.update();
        DelayedOperation op = x.set(x.get() + 1);
        op.await();
        updated = op.isOk();
    }
}
```

- Variables also support ChangeListeners, which are notified whenever the variable has been updated.

Because of the update and set semantics in *Same*, the race condition in Listing 2.1 does not occur, as shown in Listing 3.2.

3.2 State Model

The state of the system consists of a list of named objects. Each object has an identifier, the object content represented as a JSON value, and a revision number.

The revision number is used to prevent unintentional overwrites. A client cannot overwrite an object unless it has seen the current state of that object.

The state is used for variables stored by users and for internal variables. An internal variable has a `'.'` (period) as the prefix of its identifier. An example of an internal variable is `.participants`, which stores the full list of currently connected peers.

3.2.1 Global Revision Number

In addition to the list of objects, a global revision number is used to keep track of the most recent version of the system state. This is used after failures in the system in order to ensure that all clients have the most recent version of the state. The master simply requests the global revision numbers from all clients, and the client with the highest such number has the most recent state.

3.2.2 Consistency

The consistency model used by *Same* is the following.

Immutable Objects An update replaces an immutable object with another immutable object. No "incomplete," intermediary object is seen by the client when this is performed.

Per-Object Sequential Consistency Updates to a variable occur in the same order on all devices in the system. For performance reasons, sequential consistency is only on a per-object level.

Because each update has an unique identifier, this implies that all the updates that are ever seen by the system are unique, i.e. all

$$(\text{identifier}, \text{object}, \text{revision})$$

triples observed by any device in the system are distinct.

3.3 System Overview

Because *Same* is a distributed system, the hardest development challenges are related to concurrency. Several devices should run the system in parallel in a consistent fashion and manage failure conditions transparently when possible. Each device runs a program that is split into several components that run in parallel.

Here the system is described under normal operation with no errors. In Section 3.4 the error states and transitions are described.

The *Same* system consists of a variable number of devices that communicate with each other. Figure 3.1 is an example of a *Same* system with three devices. Each of the devices in the system has a client service. Client applications interact with *Same* using their local client service. The client services communicate with a master service. The master service is hosted on a single device and handles updates. The Paxos service is idle until it is needed for master election.

Note that client applications communicate with the local client service, even if their device has the master service. This is done in order to avoid code duplication.

Every device has a copy of the system state described in Section 3.2. The system state is consistent across all the devices, but each instance may not be up-to-date.

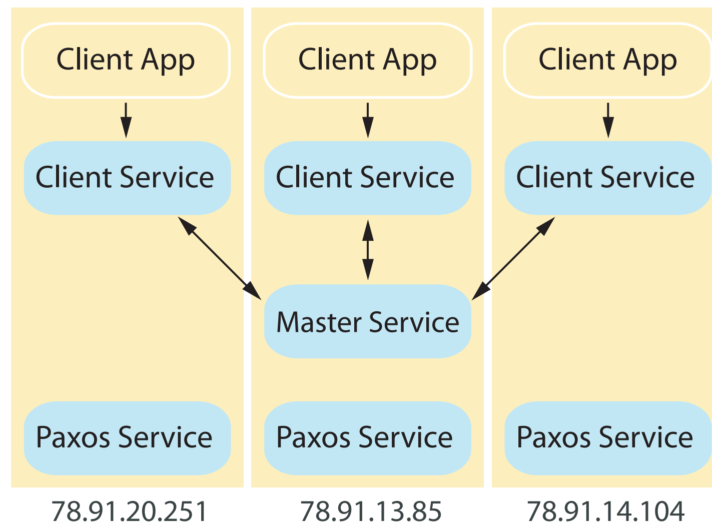


Figure 3.1 – System Overview: A system with three devices. Each device has a ClientService and a PaxosService, but only one has a MasterService. An arrow from A to B means that A interacts with B.

3.3.1 Updating State

When a client requests a change in the system state an update message is sent to the master. The update message contains the new object for a particular variable. If the client's state was sufficiently up-to-date the master accepts the update.

What is meant here by *sufficiently up-to-date* is the following. Recall that the system state consists of a set of objects with identifiers, revision numbers and object, i.e. a triple

$$(\text{identifier}, \text{object}, \text{revision}).$$

If a client requests an update of (i, o, r) to (i, o', r'), the client sends (i, o', r) to the server in an update request. The server looks up its version of the object, say (i, o_0, r_0). If $r = r_0$ then the client's version was *sufficiently up-to-date*. Equivalently, a client may update a variable v if it has the most recent version of that variable. The update will succeed even if other variables than v have been updated more recently.

3.3.2 Peer Discovery and Participation

The *Same* system is meant to discover peers automatically. Because it is assumed to run on a LAN, UDP broadcasts were used at first. Every participant in the system would listen for broadcasts on a predetermined port. When a new client wanted to connect to the system, it would broadcast its address. The participants listening for broadcasts, would then contact the source address of the broadcast with information about their network. The new client could then request to join the network.

This method of discovery was assumed to work well in most simple LANs, but *Same* was developed at NTNU using the Eduroam[3] network. This network does not support UDP broadcasts (and even if it did, devices frequently get addresses from different subnets).

Instead of using automatic peer discovery by UDP broadcasts, a centralized server was developed. When a network is created, the master sends a message to the central server containing the local IP address of the master. This address is often a local address not accessible from the internet. Clients wanting to join the system queries the server for available networks, and request permission to join the network.

Many issues arose because of the centralized server. The system was supposed to be used on LANs, but because of this server the systems are globally advertised. We can imagine means to solve these problems, but this has not been a focus area of this project.

3.4 System State

The *Same* system has four possible states. The current state of the network determines the behavior of the participants and user interaction with the system.

These states are maintained by the clients. All of them may not have the correct view on the system at all times. For instance, if a client loses connection to the system, the master will drop it on the next update. From the master's point of view, the client is disconnected, but from the client's point of view it is still connected to the network, until the next time it tries to run an update which will fail.

Disconnected The client is not currently connected to a network. The system is idle, and user interaction results in errors. When the client connects to a network, it enters normal operation.

[3] http://www.eduroam.org

Normal Operation A set of clients are connected to a master. Variables are updated and broadcasted to the entire set of clients. If a client fails, the client goes to the *no master* state. The master drops the participant and the remaining peers continue in *normal operation*. If a master fails, the master is disconnected, and the clients enter the *no master* state.

Some of the clients may fail to be notified of a master failure. They will either eventually go into the *no master* state or be notified when a new master takes over.

No Master When a client is in a *no master* state, it attempts to establish a *consensus* on which client should become the new master. If more than half of the current participant list is in this state, they will elect a new master and assume *normal operation*. Otherwise, master election will fail and all clients in the *no master* state will be *disconnected*.

If the client application attempts to set a variable in this state, the caller thread is blocked until the client enters *normal operation*.

Master Takeover Period After a master has been elected, all the participants of the network need to acknowledge the new master and the system has to ensure that all clients have a consistent view of the current state.

Inconsistency may have emerged because clients left the *normal operation* at different times, e.g., because some clients lost their connection to the master before others. If the remaining clients in *normal operation* managed to perform state updates, the clients in the *no master* state would not know about these updates.

All the clients which acknowledge the new master enter *normal operation*. Other clients are dropped the network and enter the *disconnected* state.

User interaction in the *master takeover period* is the same as in the *no master* state.

3.5 Participant Overview

The participants are responsible for maintaining their own state, and their view of the global system according to the network protocol. Additionally they provide a programming interface to allow interaction with the system.

Listing 3.3 – Service specifications. Each service has a set of methods that take one message parameter and returns another. A "void" method is one that returns an empty message.

```
service Client {
    rpc SetState (Component) returns (Empty);
    rpc MasterDown (MasterState) returns (Empty);
    rpc MasterTakeover (MasterState)
        returns (MasterTakeoverResponse);
    rpc GetFullState (Empty) returns (FullStateResponse);
    rpc MasterTakeoverFinished (MasterState) returns (Empty);
}

service Master {
    rpc JoinNetworkRequest (ClientState) returns (Empty);
    rpc UpdateStateRequest (Component)
        returns (UpdateComponentResponse);
}

service Directory {
    rpc RegisterNetwork (MasterState) returns (Empty);
    rpc GetNetworks (Empty) returns (NetworkDirectory);
}

service Paxos {
    rpc Propose (PaxosRequest) returns (PaxosResponse);
    rpc AcceptRequest (PaxosRequest)
        returns (PaxosResponse);
}
```

3.5.1 Services

Each participant runs a set of services, served using Simple-Protobuf-RPC (Section 4.4). A service handles remote procedure calls from other participants. The services are specified using the protocol buffer language[4]. The definitions of the main services used in *Same* are shown in Listing 3.3 and the message definitions in Listing 3.4. These listings show protocol buffer code. They are compiled into Java code using protocol buffer compiler.

[4] http://code.google.com/p/protobuf/

Listing 3.4 – Some of the message types used in Same.

```
message Empty {
}

message UpdateComponentResponse {
    required bool success = 1;
}

message Component {
    required string id = 1;
    required string data = 2;
    required int64 revision = 3;
}

message MasterState {
    optional string master_url = 1;
    optional int32 master_id = 2;
    optional string network_name = 3;
    optional string master_location = 4;
    optional int64 revision = 5;
}

message MasterTakeoverResponse {
    optional bool success = 2;
    optional ClientState client_state = 3;
}

message FullStateResponse {
    optional int64 revision = 1;
    repeated Component component = 2;
}

message ClientState {
    optional string url = 1;
    optional string location = 2;
    optional int64 revision = 3;
}

message NetworkDirectory {
    repeated MasterState network = 1;
}

message PaxosRequest {
    optional ClientState client = 1;
    optional int32 proposalNumber = 2;
}

message PaxosResponse {
    optional int32 result = 1;
}
```

3.5.2 Master

The *Master* service governs the global state of the system. The global state consists of the user state as explained as the State Model, the set of participants, and other meta data about the system. When a client wants to join or update an object, it contacts the master service, which handles the request.

When a master has failed and another master takes over, the new master has to resume the operations of the system. The master handles recovery of the state during the *master takeover period* described in Section 3.4.

3.5.3 Client

The Client is responsible for a number of tasks to a participant:

Client Service Receive state updates from the master and apply them to the local state copy.

Programmer Interface Handle asynchronous state operations from the client program. State updates from the client program must be sent to the master. State updates from the master must be relayed to the appropriate program objects.

Master Election When the master fails, the client starts a master election by sending a MasterDown() message to the other clients. This process is described in Section 3.6.

3.5.4 Paxos

The *Paxos* service is a fairly standard realization of the *Paxos* protocol. Its implementation is discussed in Section 4.6.

3.5.5 Directory

The *Directory* service is used as a registry of available networks. When a network is created, it is registered in this service using the RegisterNetwork() method. Another device may use the GetNetworks() method in order to get a list of active networks.

Networks automatically expire after a certain time. In order to keep the network in the directory, the devices must re-register the network before it expires.

3.5.6 SystemService

The *SystemService* (not shown in the previous listing) is used for testing. The GetSystemStatus() method returns information about the running components in the system. It can be used to determine if two devices have consistent state.

The KillMaster() method is used to kill the master of the network. If this method is called on the participant hosting the master service, it will kill the master immediately. Otherwise it will forward the request to the current master.

3.6 Master Takeover

When a master fails or becomes unavailable, a message from any client will fail. When this happens, the client enters the *no master* state and broadcasts a *no master* message to the rest of the network. (This message needs no error handling. If it reaches any other clients, they will join the master election, otherwise it has no effect.)

At this point at least one client start a MasterProposer. The MasterProposer contacts all the available Paxos services. The MasterProposer tries to establish a consensus value. It will either succeed with some proposal n, or it may be aborted if some other MasterProposer finished and a new master was already elected.

When a MasterProposer succeeds, it will start a new master with id n (from the proposal number). The master sends a MasterTakeover() message containing the master id to all the clients, and the network enters the master takeover period.

When a client is notified of a new master, it accepts it if its id is greater than the current master. If several masters were activated with different ids (which is possible or even likely, due to the behavior of the Paxos protocol), one master will have the highest id, and all clients will accept it as the new master. The new master resolves possible inconsistency in the master takeover period, after which the network continues in *normal operation*.

An example of this process is illustrated in Figure 3.2.

3.6.1 Master Election Algorithm

Paxos has been used as the master election algorithm in *Same*. This choice was made arbitrarily.

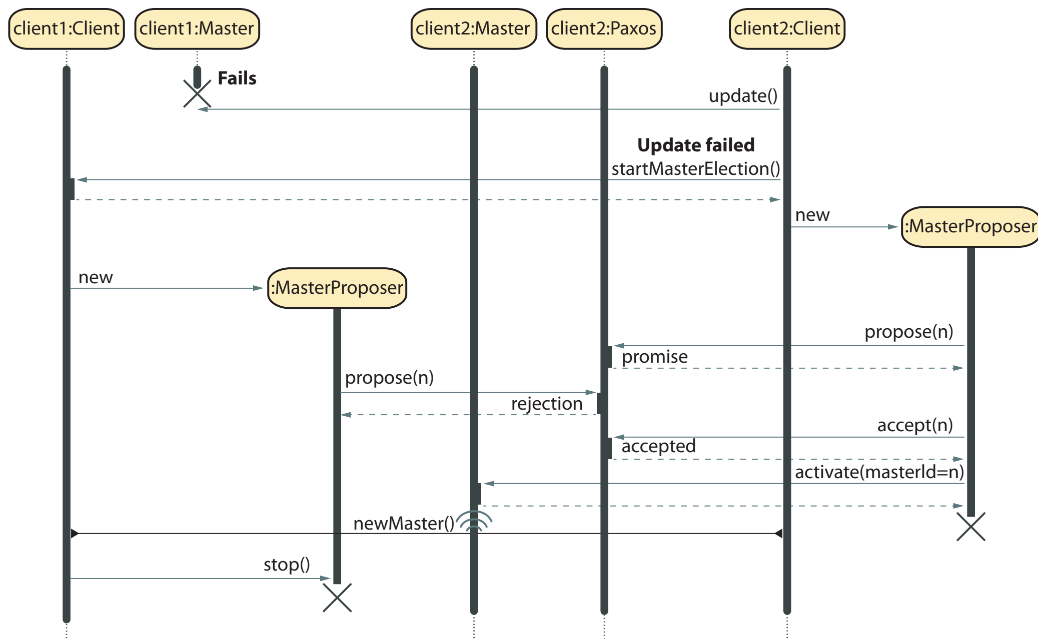


Figure 3.2 – Master takeover: This diagram is a visualization of a master takeover that happened in a Same system running on two mobile devices. The log messages from the devices were translated into this sequence diagram.

Master1 and client1 run on device A. The rest of the objects run on device B. After master1 fails and client2 attempts to update a variable, the master takeover starts. Each of the devices start a MasterProposer. Device B establishes a consensus with the only available Paxos service, paxos2 and becomes the new master.

The *Same* networks that were tested during development consisted of devices with very different performance. If a low performing device becomes the master it greatly affects the performance of the entire system.

If the Bully Algorithm was used for master election, the devices could run a benchmark to assess their performance and use this measurement in the master election. Then the best performing device would be the master with a high probability. Due to the problems with a low performing master it was probably a mistake to use Paxos.

3.6.2 Available Paxos Services

Paxos guarantees consensus when a proposal has been accepted by a majority of the Paxos servers in the system. This means that a proposer must know the total number of servers in order to determine whether it achieved consensus.

In *Same* the current participants are broadcasted to the entire system. When a master fails the list of participants is used as a basis for the available Paxos services, but the Paxos service from the failed master is removed from the list. A master may fail for various reasons, and in some cases the device may still be available. If it is available, it is possible to utilize its Paxos service. Otherwise, using its Paxos service may result in an unrecoverable state: Consider two devices, A and B. Device A is the current master, which fails. Device B will attempt to establish a consensus value by proposing a value to A and B. If A's Paxos service is unavailable, B will get only 50 percent acceptance, which is not a majority. Excluding A's Paxos service from the master election solves this problem.

3.6.3 Master Recovery

After a master has been elected, the master takeover period starts. At this point, there may be inconsistencies in each client's state. The master restores consistency with the following steps.

1. Send master takeover message. This aborts other MasterProposers.
2. Retrieve the most recent state. This is done by requesting the global revision number from each client. If any client returns a revision number that is higher than the master's revision number, the master requests the state from that client.
3. Send the updated state to all clients. *Same* sends the full state to all participants, even though the master may use the revision numbers to

determine exactly which state it must send. This optimization may speed up the master takeover time if many large objects are stored.

4. Any clients that failed to receive any of the previous messages sent during the master takeover period are removed from the network. This is done because otherwise those clients may have state that is inconsistent with the rest of the system.
5. Send a takeover finished message. All clients resume normal operation.

Chapter 4

Implementation

Same is designed as a Java library that can run on a Linux computer and on Android mobile devices. The basic implementation and testing was done for a Linux computer. The system was adapted for the Android operating system.

During the development of *Same*, many bugs related to concurrency were observed, and this chapter explains some of the experiences. In addition, the RPC system *Protobuf-simple-RPC* is presented.

4.1 Concurrency Model

In a distributed system, many operations run in parallel at any time. Even a single participant runs many concurrent operations, e.g., issuing asynchronous state updates and processing requests.

Programming concurrent applications is very hard, especially when directly using the concurrency primitives offered by Java. For instance, forgetting to notify blocked threads may lead to deadlocks while forgetting to synchronize a code block may lead to inconsistent state.

Using Java's `java.util.concurrent` library may make our programs less prone to some of these errors. For instance, using atomic variables and thread-safe data structures removes the need for manual synchronization. However, other errors may occur. Producer-consumer queues and executors lets us build concurrent components that do not rely on internal locking. However, unless it is used correctly, it may introduce thread starvation.

4.1.1 Producer-Consumer Queues

Early versions of *Same* used concurrency primitives directly, but this caused several bugs in the master and client code. Classes were rewritten to using producer-consumer queues, resulting in less code and fewer bugs.

The queues are typically used for delayed commands. Most of the service objects hold a lock while processing an incoming query. If such an object sends an RPC to itself it could cause a deadlock. For instance, when the master receives a state update it should respond immediately instead of notifying other services of the update while blocking the master thread.

Late in its development, *Same* switched to an asynchronous RPC implementation. This removes the issue of deadlocks, because threads typically do not block while waiting for an RPC to finish. Instead, encoding the wrong set of dependencies in RPCs may cause thread starvation, which is discussed later.

4.1.2 Message Passing

An Android application is organized as *Activities* and *Services*. An Activity is an object with a user interfaces that the user interacts with directly. A Service is an object that performs tasks in the background without user interaction. The Android platform supports different forms of message between different objects, and between different processes.

Same is implemented as a server application, maintaining its own state and communicating with other participants in the background. Therefore it runs as a Service on Android. Any Activity that uses *Same* communicates with the *Same* service using a message passing interface.

The Service model clearly separates persistent components from components that may be garbage collected by the operating system. Whenever an Activity is not visible to the user, Android may shut it down in order to conserve system resources.

Unfortunately the message passing interfaces used in Android is verbose and not very easy to use. One goal of *Same* is to provide a simple programmer interface. To achieve this, the *Same* programming interface was replicated on Android. The `ClientInterfaceBridge` class implements the same programming interface, but communicates with the *Same* service using message passing.

Android's message passing is handled by a single main thread (similar to the event-dispatch thread in Java Swing). Blocking this thread can have unfortunate consequences. In *Same* an object sent a message to another object and then tried to block the thread until a response was received. Because the message handling thread is blocked, the target object could never process its message, resulting

in freezing the entire program. Strictly speaking this is not a concurrency bug, because only one thread was involved, however it is very closely related to thread pool starvation.

4.2 Thread Pool Starvation

In early versions of *Same*, the different services could be executed in parallel. Recall that when the master fails, clients receive a MasterDown() request. When clients got this message, they used to send propose requests to the Paxos services directly.

After switching to asynchronous RPCs the clients would freeze when trying to elect a new master. The reason for this is that requests are scheduled in a single-thread thread pool. When a client sends a request to the Paxos service, it is blocked until it receives a response. Unfortunately this blocks the only thread in the thread pool and therefore the Paxos request can never be processed. This freezes the client, but more severely the RPC server cannot process any requests, not even to the debugging service. This bug was located by inspecting the stack traces of the running threads.

4.3 Remote Procedure Calls

Same uses Remote Procedure Calls (RPC) for all its communication. Although RPCs are widely used, the majority of the available libraries are targeted to server systems. Several RPC implementations were evaluated for use in *Same*.

- Java RMI is the standard RPC library in Java. It is not supported on Android.

- Apache XML-RPC[1] is an RPC library that uses XML over HTTP. An attempt was made to use this library with *Same*. Unfortunately the XML library used internally by XML-RPC is not supported on Android.

- JSON-RPC for Java[2] (*jsonrpc4j*) is similar to XML-RPC, but uses JSON instead of XML. It can be used directly with Java interfaces, which makes it very easy to use this RPC mechanism, as well as removing the transport layer under test.

[1] http://ws.apache.org/xmlrpc/
[2] http://code.google.com/p/jsonrpc4j/

Initially, *Same* adopted *jsonrpc4j* using the *Jetty* server as the transport layer. Unfortunately a bug in the HttpUrlConnection class in Android versions prior to 2.3 made it impossible to use *jsonrpc4j* with persistent connections [3]. This was fixed by implementing a custom transport layer using the *Apache HTTP Client*[4].

The *Jetty* web server is a full-fledged web server for Java that can be embedded in applications. A web server is a useful tool for monitoring and debugging of distributed applications. A special servlet was written to allow interaction with the system through a web browser.

Because Android phones have limited computing resources, *jsonrpc4j* over HTTP provided by *Jetty* is quite slow. On older models such as the HTC Magic the RPC latency was not satisfactory for real-time applications.

4.4 Toward a Faster RPC for Mobile Devices

The RPC performance with some of the existing solutions is not good enough for real-time applications. Potential reasons that *jsonrpc4j* over Jetty is too slow on Android devices may include

1. Jetty is a full-featured web server not optimized for the relatively low-end mobile devices used in this project. For instance, Jetty may work more efficiently with a larger amount of RAM and may have code that runs faster on JVMs other than Dalvik.

2. Usage of inefficient formats. HTTP requests need to be parsed and interpreted by Jetty, and HTTP has features that are irrelevant to this application. JSON is a human-readable format that needs to be generated and parsed with each request.

3. *Jsonrpc4j* uses the Java reflection API and so does the JSON library Jackson[5] which is used internally. This may give worse performance compared to compiled Java code.

A lightweight specialized RPC solution for mobile devices may address all of these issues. Protocol Buffer[6] is a binary format written to be platform-independent, fast to serialize and unserialize and have a much smaller binary

[3] http://code.google.com/p/android/issues/detail?id=2939
[4] http://hc.apache.org/httpcomponents-client-ga/
[5] http://jackson.codehaus.org/
[6] http://code.google.com/p/protobuf/

footprint than formats such as JSON and XML. Protocol Buffers do not include an RPC mechanism directly, but provides services that can be used for RPC with a specialized transport layer.

Protobuf-simple-RPC[7] was developed for this project as an attempt to create a fast RPC for mobile applications. In addition to overcome the issues mentioned above, it provides a configurable asynchronous RPC mechanism.

It is difficult to implement a concurrent server. Because it was supposed to have good performance on mobile devices, an architecture based on the Java non-blocking I/O libraries (java.nio) could handle several connections from a single thread. This was not done because java.nio is harder to use than blocking sockets, especially when using protocol buffers[8]. Additionally, an asynchronous interface is easier to implement with multithreading.

For these reasons, *Protobuf-simple-RPC* uses the blocking sockets in Java. It is meant to handle a relatively small number of concurrent connections and in fact uses several threads per connection in order to handle requests independently and asynchronously. It uses producer-consumer queues to communicate between threads. For instance, one thread writes responses to a socket from a blocking queue. A single connection may have an arbitrary number of ongoing requests. Whenever a request is processed the response is simply pushed to the queue. The number of concurrent requests is bounded by the queue and thread pool sizes, which are configurable. An overview of this design is illustrated in Figure 4.1.

Sequential execution of queries is not guaranteed by *protobuf-simple-RPC*. Requests are asynchronous and may be scheduled in any order by the server. If sequential execution is required, this may be achieved by setting the request handling pool size to one. This pool is shared by all the services running in a server instance, and therefore guarantees sequential consistency. On the client side, each request is atomically added to a blocking queue and sent in order. Even though the server processes requests synchronously, several requests may be queued and sent over the network in the same TCP packet, giving a performance boost. This is the mode that is used in *Same*. All the services used by *Same* were initially made for a synchronous RPC library. A result of this was that the request handlers had to respond quickly in order to minimize the latency of the synchronous calls. If the response handlers execute fast, the performance gain of concurrent request handling may be small.

[7] https://github.com/orbekk/protobuf-simple-rpc

[8] Protocol Buffers provide a mechanism to interact with blocking streams. If java.nio were to be used, messages would have to be buffered and assembled manually.

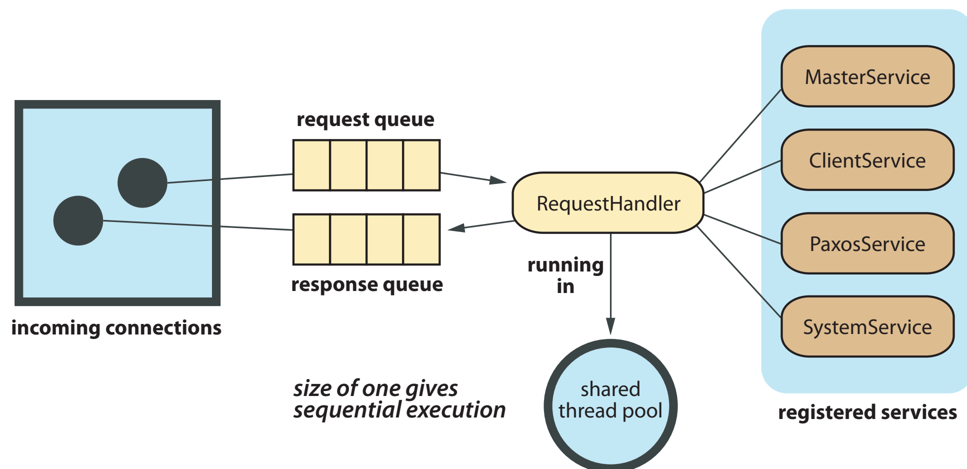


Figure 4.1 – The Protobuf RPC server. Incoming connections are handled by dedicated threads (running in a thread pool not shown in this diagram) that read queries and push them to the request queue. RequestHandlers take queries from the request queue and executes methods in one of the registered services. When a method completes, its response is pushed to a response queue and later written back to its respective connection.

Most of the server components may be parallelized. All the yellow components are per-connection. This includes the queues (which are also bounded). If one connection is slower than others it should not slow down the speed of the system, because the other connections run in separate threads. However, if the pool size of the shared thread pool is set to one the server handles all requests sequentially.

Table 4.1 – Requests per second processed by mobile devices.

RPC implementation	HTC Magic	Samsung Galaxy S
Jsonrpc4j+Jetty	31	85
Protobuf-simple-RPC	94	884

Table 4.2 – Requests per second sent and processed by mobile devices.

RPC implementation	HTC Magic	Samsung Galaxy S
Jsonrpc4j+Jetty	16	69
Protobuf-simple-RPC	66	570

4.5 RPC Performance

In order to evaluate *protobuf-simple-RPC* a benchmark was performed comparing it to *jsonrpc4j* with *Jetty*. Similar services were programmed for *jsonrpc4j* and *protobuf-simple-RPC*, in which a simple function is computed and returned by the server.

These benchmarks are attempts to measure the performance of the services under normal operation. Each benchmark consists of a warm-up round and the main benchmark. The warm-up round forces the devices to run slow startup code such as loading Java classes.

4.5.1 Benchmark 1

In this benchmark requests were sent from a Linux PC to mobile devices over a consumer-level Wi-Fi network. This is an attempt to measure the end-to-end performance of the TCP stack and the RPC implementation on these devices. The results are shown in table 4.1.

4.5.2 Benchmark 2

In this benchmark, the mobile devices sent requests to themselves. This eliminates network latency but uses operating system sockets and therefore the performance of the TCP stack could still be a factor. The results are shown in table 4.2.

4.5.3 Evaluation

Protobuf-simple-RPC has much better performance than *jsonrpc4j* over *Jetty* in these measurements. On the Galaxy S, performance is up to 10 times faster with *protobuf-simple-RPC*. Interestingly, an increase of only a factor of 3 is seen on HTC Magic.

If more effort were to be put into *protobuf-simple-rpc*, it may be possible to achieve even better results. For instance, if the usage of a high number of threads is a performance bottleneck, an alternative implementation could use the `java.nio` library and Android's internal message passing interface. This implementation could run in a single thread.

A Note on Object Pooling

Object pools are occasionally used on Android. For instance the `Message.obtain()`[9] method manages a pool of messages for performance reasons. An official Android performance guide advises against creating unnecessary objects[10].

Protobuf-simple-rpc creates a handful of objects for each request. In a benchmark that executes thousands of requests, one may think that object creation and garbage collection could affect the performance. One benchmark was attempted for *protobuf-simple-RPC* with a single recycled object instead of new objects for each request. This gave no performance increase, and confirms that object pools should not be used without measuring their effect on performance.

4.6 Paxos Implementation

As an example of a concurrent component, a discussion of *Same*'s Paxos implementation is presented.

4.6.1 Paxos Service

The Paxos service itself was very easy to implement: The only thing it needs to do is to perform the acceptor algorithm, which is simple. The implementation used here is based on the original algorithm[4] but sends reject messages. The service may be tested by setting internal state (the highest promised and

[9] http://developer.android.com/reference/android/os/Message.html
[10] http://developer.android.com/guide/practices/design/performance.html

accepted values) and testing the responses of the service (which are completely deterministic). Because clients may contact the service concurrently, it should be thread-safe. This was accomplished with a lock.

4.6.2 Paxos Client

The Paxos client was harder to implement. When it sends queries to a set of Paxos servers, it must handle the responses concurrently and proceed when enough servers accept. This was implemented using components from the Java concurrency library[11]. The client makes use of atomic variables instead of locks, and a CountDownLatch as a barrier. The ResponseHandler in Listing 4.1 is the callback used with the RPC requests, and handles the synchronization used in the Paxos client. An example of its usage is given in the propose() method in Listing 4.2.

ResponseHandler completes immediately if enough promises have been received, but waits longer than strictly necessary otherwise (it could fail immediately when a majority of the requests fail). This was done by design because if there is a failure, the Paxos client will wait a while before retrying, meaning that optimizing this case is unnecessary. Additionally, a Paxos server may respond late with a higher promised value – waiting until all the requests have been received ensures that result is not changed to a different value after it has been set[12]. Trading a dubious optimization for predictable concurrent behavior was presumed to be a good decision.

Experiences with Lock-Free Objects

Using lock-free techniques as in ResponseHandler seemed simple at first: All the variables are updated atomically and new values are visible to all threads immediately. However, ResponseHandler has several atomic variables and after careful analysis of the thread safety of checkDone() a bug was discovered. If numResponses is updated before numPromises (this was the case in an earlier version) there is a race condition that makes it possible for ResponseHandler to incorrectly report a failure (much better than incorrectly reporting a success). This race condition is very unlikely to occur in practice[13], and only in special

[11] java.util.concurrent

[12] This relies on numRequests being set correctly according to the number of RPC requests. Then numResponses.get() $\geq$ numRequests only if they are equal.

[13] In *Same* it would never occur, because the RPC server uses only a single thread to execute callbacks. *Same* deliberately relies on this for sequential consistency, but the Paxos client was not meant to rely on it.

circumstances that depend on the responses of the Paxos servers, and it would probably result in the Paxos client to fail only slightly more than it should. This is no means a problem, but it illustrates how difficult concurrency can be.

4.7 Dependency Injection

Same uses the dependency injection pattern throughout the code. Initially this was in order to write efficient unit tests and also due to personal preference. At a later time in the development it proved to be useful in this project for several other reasons as well.

- The implementation of UDP broadcast had to be changed for the Android platform. A custom implementation for Android could be substituted for the Java implementation because the components were loosely coupled.
- Very poor performance was observed on old mobile devices. The Jetty web server was a suspected reason for this. Because the code was loosely coupled and used the HttpServlet technology, Jetty server could easily be substituted with another web server. Performance was tested using Tiny Java Web Server, but this did not increase the performance.
- Testing remote procedure calls. *Same* uses RPC to interact with the other components in the system. With dependency injection, local versions of the services can be used in unit tests. Instead of sending RPCs over a network connection, RPCs are sent directly to a local object.

Listing 4.1 – The ResponseHandler callback used by the Paxos client.

```
class ResponseHandler implements RpcCallback<PaxosResponse> {
    final int proposalNumber;
    final int numRequests;
    /** Contains -n, where n is the highest accepted proposal. */
    final AtomicInteger bestPromise = new AtomicInteger();
    final AtomicInteger numPromises = new AtomicInteger(0);
    final AtomicInteger numResponses = new AtomicInteger(0);
    final AtomicInteger result = new AtomicInteger();
    final CountDownLatch done = new CountDownLatch(1);

    public ResponseHandler(int proposalNumber, int numRequests) {
        this.proposalNumber = proposalNumber;
        this.numRequests = numRequests;
        bestPromise.set(-proposalNumber);
    }

    /** This method is called when an RPC completes. */
    @Override public void run(PaxosResponse response) {
        if (response != null) {
            int result = response.getResult();
            if (result == proposalNumber) {
                numPromises.incrementAndGet();
            }
            boolean updated = false;
            while (!updated) {  // Lock-free update of bestPromise.
                int oldVal = bestPromise.get();
                int update = Math.min(oldVal, result);
                updated = bestPromise.compareAndSet(oldVal, update);
            }
        }
        numResponses.incrementAndGet();
        checkDone();
    }

    private void checkDone() {
        if (numPromises.get() > numRequests / 2 ||
                numResponses.get() >= numRequests) {
            if (numPromises.get() > numRequests / 2) {
                result.set(proposalNumber);
            } else {
                result.set(bestPromise.get());
            }
            done.countDown();
        }
    }

    public int getResult() throws InterruptedException {
        done.await();
        return result.get();
    }
}
```

Listing 4.2 – The ResponseHandler callback used by the Paxos client.

```
/** Propose a value to the paxos servers.
 * If the proposal is accepted, returns proposalNumber.
 * Otherwise it returns -n where n is the highest promise
 * that has been observed. */
private int propose(int proposalNumber)
        throws InterruptedException {
    ResponseHandler handler = new ResponseHandler(proposalNumber,
            paxosLocations.size());
    for (String location : paxosLocations) {
        // Get a connection to the paxos service.
        Services.Paxos paxos = connections.getPaxos(location);
        if (paxos == null) {
            // The paxos service at 'location' is not reachable. This
            // counts as one failure.
            handler.run(null);
            continue;
        }
        Rpc rpc = rpcFactory.create();
        PaxosRequest request = PaxosRequest.newBuilder()
                .setClient(client)
                .setProposalNumber(proposalNumber)
                .build();
        // Send the proposal; 'handler' is the callback.
        paxos.propose(rpc, request, handler);
    }
    // This blocks until enough servers responded or requests
    // timed out.
    return handler.getResult();
}
```

Chapter 5

Evaluation and Experiences

This chapter discusses experiences gained while using *Same*. An example application with shared state has been developed and benchmarks were performed. The benchmarks target specific requirements.

5.1 Example Application

The example application draws a movable circle on the screen. When the user touches a point, the circle moves there. The application is shown in Figure 5.1. The circle is represented by a *Same* object containing its position.

When several devices are connected and run the example application, the location of the circle is shared. Touching any of the devices moves the circle on all the connected devices.

A simple implementation of this is to change the variable whenever the user touches the screen. However, *Same* does not support changing a variable twice unless the first change is allowed to finish before sending the second. Additionally the application receives too many touch events for this to be efficient.

A better approach is to remember the most recent touch event and update the variable when the variable is ready to get updated *or* when another touch event has been received. This was harder than it may appear for the following reasons.

- It is necessary to synchronize on several conditions: Touch events and "ready to update". The latter is true when we have the most recent version of the variable.

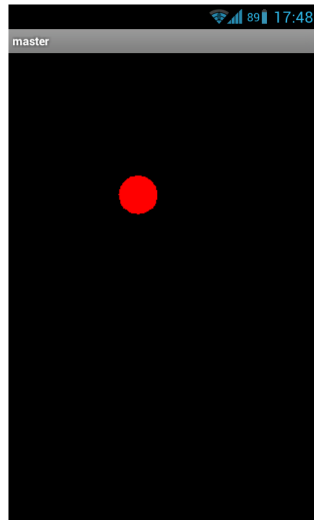


Figure 5.1 – The Example Application running on one device.

- Errors have to be handled. An update request may give an error if there is a conflict, if the local client lost connection to the master, etc. Some errors should result in retries, others should abort the update.

- The touch events from the user and update events from *Same* are received in the UI thread. Blocking this thread may result in an unresponsive application or in the worst case thread starvation. Therefore blocking operations have to be performed in a separate thread.

These issues are common for all applications that want to use *Same* to continuously update a variable based on user input. The class VariableUpdaterTask handles this type of update pattern and is included in *Same*.

A video demonstration of the example application may be found at `http://youtu.be/XOF6oeEPJNY`.

5.2 Benchmarks

All benchmarks have been performed on Android devices on the Eduroam Wi-Fi network at NTNU. The devices run a few warm-up iterations and then collect results from a number of benchmark iterations. The graphs show average values and the standard deviation of the samples. An overview of the specifications of the devices is given in Appendix B.

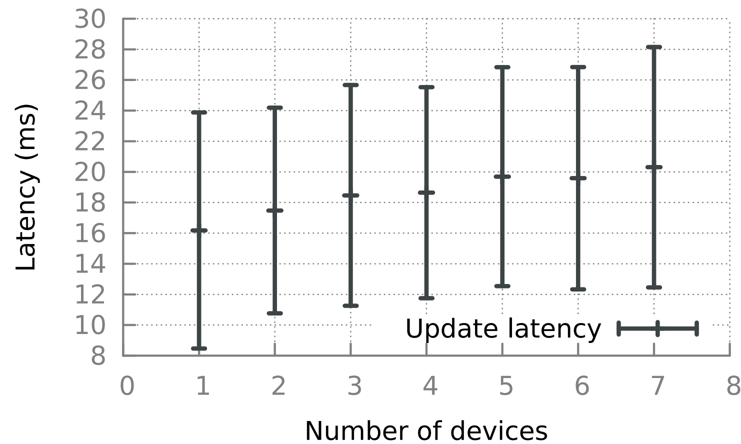


Figure 5.2 – Update latencies in benchmark 1. The benchmark performs 2000 variable updates after 100 warm-up iterations.

5.2.1 Benchmark 1: Update Latency

This benchmarks measures end-to-end latency of updates. One device updates a variable a number of times and measures the time until the master sends the updated version of the variable. This is repeated with different numbers of devices in the network.

The latency of an update is important in order to support real-time applications. Low latency was one of the requirements according to Section 1.4. The update pattern in this benchmark is similar to the behavior in the example application.

The result is shown in Figure 5.2. There is very little increase in the latency as new devices are added. The results from this benchmark are very stable, even though the test was performed with a variety of devices. The reason for this is that the bottleneck is the performance of the master device. As long as all the clients can keep up with the updates from the master, the results are dependent on the performance of only one device.

5.2.2 Benchmark 2: Master Takeover Time

Whenever the master fails in *Same*, there is a period when no updates can be performed in the system. A real-time application may freeze during this period, which is what happens in the example application[1]. If the master can take over in a short period of time, this has less negative impact on the application.

[1] This can be observed in the video demonstration.

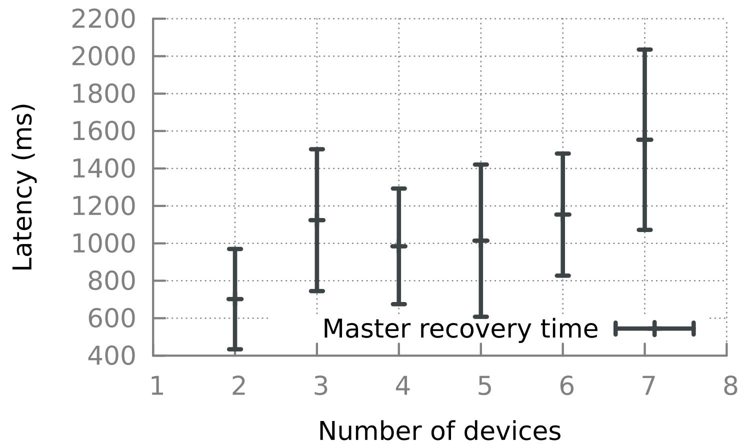


Figure 5.3 – Master takeover time average in benchmark 2. This benchmark kills the master service and measures the master takeover time 50 times after 2 warm-up iterations.

This benchmark measures the end-to-end recovery time after the master fails. A number of devices participate in a network, and then the master is killed repeatedly. After the master is killed, one of the participants attempts to update a value and measures the time until the update can be performed. This experiment is repeated with a varying number of devices and.

The result of this benchmark is shown in Figure 5.3. It appears that the master recovery time increases as the number of devices increase. Apart from that, the graph looks rather strange. This benchmark has a few problems that are discussed below, and therefore we should not completely trust the results.

Few Iterations This benchmark was performed with only 50 iterations. We had limited access to test devices, and because each iteration of this benchmark took long to run, the number of iterations were decreased.

Randomized Process The Paxos algorithm is randomized, i.e., it is random whether it finishes after one iteration or if several iterations must be performed. In addition, the master proposers have a randomized back-off timeout. This process may have surprising running times based on the number of devices.

Different Masters In Benchmark 1, one device was allowed to remain the master during the entire test. In this benchmark, the master is elected randomly among the participants. Unfortunately the benchmark was executed on five different types of devices. Because the master has a large

impact on the performance of the network, this may have impacted the result of the benchmark.

5.3 Development Environment

For the curious reader this section is a summary of the development environment used in this project and the experiences with Android development and testing with multiple devices.

5.3.1 Software

The Eclipse IDE was used for the Java programming. The Android plugin for Eclipse includes a debugging environment that makes it possible to debug running processes on Android devices.

The Android Debug Bridge (adb) daemon automatically connects to all devices that is plugged into the system. No special setup need to be performed in order to develop on multiple devices.

Apache Maven was used for dependency management, building and execution. It can be set up for an Android project to build, install and run an Android application on all connected devices. This was done in order to run and test the programs in this project.

The adb command line tool was used to examine logs. For each device, one terminal window with adb logcat running gives real-time log messages from all the devices in the system.

5.3.2 Benchmarking

In order to perform repeated experiments it was necessary to create a framework to collect results from benchmarks. Each benchmark is an Android Activity that should run and measure performance. The measurements should be collected to a single location and combined into one data file.

This project already uses RPCs and multiple services. Because the infrastructure was already in place the natural solution was to create a benchmark server. The benchmark server hosts a special service for each benchmark. When the benchmark Activities run, they connect to the benchmark server in order to report their results. The benchmark server made it easy to repeat experiments with different parameters.

5.3.3 Debugging

The main debugging tool was log messages written by the system. Looking at logs in real-time lets us observe whether commands are transported correctly between the devices and in which order events occur. However, the logs can get very verbose, and too much logging may slow down the application. *Same* uses the *slf4j* logging framework that allows configuration of logging of different parts of the system. For instance, logging may be selectively enabled for the RPC library. In that case all incoming and outgoing RPCs from the system are printed, which is great for debugging but slows down the system drastically.

A special service called SystemService was developed to inspect and control the state of a device. The SystemService.GetSystemStatus() method can be called to get information about the state of the client and master components of a system.

As mentioned, the Android Eclipse plugin was used to debug the application to some extent. One useful usage of this was to inspect stack traces that uncovered a thread pool starvation bug.

5.3.4 Automatic Testing

Same was programmed with standard Java without using any Android classes for the main functionality. This allowed tests to be written in standard Java as well. The testing frameworks *JUnit 4*[2] and *Mockito*[3] were used to create unit tests and functional tests. Testing that individual components behave correctly in isolation increases the chance that the components work well together.

5.4 Evaluation

This section is an evaluation of *Same* based on the requirements in Section 1.4.

Localized Architecture It has a mostly localized architecture. The exception to this is the directory service that is used for peer discovery. As mentioned in Section 3.3.2 there may be some ways around this, but at this time we do not have a satisfactory solution.

In the spirit of this requirement, all *Same* operations run on a local network. The discovery service is only used for peer discovery.

[2] `http://www.junit.org`

[3] `http://code.google.com/p/mockito/`

Low Latency *Same* has reasonably low latency. It may be good enough for real time applications, but not with many objects transferred concurrently.

Scaling *Same* appears to scale very well. Only a small increase in the update latency is seen when new devices join the network. However, with a larger network, more updates can be expected. In that case the performance may not be good enough, but this is likely a latency problem and not because of poor scalability.

Accounting for Unreliable Devices When a device fails, *Same* is able to quickly choose a new master and resume normal operation.

Same should be able to handle failures under the same condition as the Paxos protocol which is used to elect the new master.

Programming Model *Same* provides a programming model that lets users share many types of Java objects using JSON serialization. The objects are synchronized and *Same* provides update semantics that are convenient for the programmer.

Same has been used to develop one example application and different test and benchmark applications. The experience suggests that *Same* is useful for game development. Furthermore, *Same* is a platform that may be extended with more features and it uses the client-server model that game developers are used to.

Discovery and Connection As previously mentioned, a directory service is used for discovery. The directory service is very simple, and is inconveniont to use. Therefore, this requirement has not been completed and is left as future work.

Overall, *Same* has fulfilled most of its original requirements and successfully provides distributed objects to Android devices. However, with more effort, *Same* could become an even better system. Possible directions and further work is discussed in Section 6.2.

Chapter 6

Conclusion and Further Work

This project presented a distributed model for sharing state for mobile multiplayer games and applications. The model is based on several assumptions about networking in multiplayer games, such as the need to share global state. Similar systems exist, but have been designed for larger systems and therefore they are not well suited for mobile applications.

This chapter gives a summary of the results of this project and possible further work with *Same* and related technology. Some lessons that were learned while developing this distributed system will be presented. Finally, this thesis concludes with remarks about *Same*.

6.1 Resulting Artifacts

This project has produced two open source libraries that anyone can use. *Same* is available at `https://github.com/orbekk/master`. The protocol buffer based RPC library that was developed for this project is available separately at `https://github.com/orbekk/protobuf-simple-rpc` and may be used by any Java project. Both of these projects are licensed under the Apache License version 2.0.

A video demonstration of *Same* is available at `http://youtu.be/XOF6oeEPJNY`. The video shows the example application and the user experience when a master recovery occurs. Another video is available at `http://youtu.be/ADtXy1Ggjvo`. This video shows the responsiveness of *Same* when running the example application with 6 devices.

6.2 Further Work

The goal of this project was to show that a distributed system can be used to share state in real-time multiplayer games. *Same* has been shown to work, but only with an example application. Some assumptions were made about how multiplayer games tend to share state, and verifying that that the model is useful is the next step for *Same*.

For that reason we would like to evaluate *Same* in a real mobile game setting, e.g. by implementing a multiplayer game. However, the scope of this master's thesis is too small to include this.

In addition, several improvements to the implementation of *Same* could increase its usefulness:

1. As previously mentioned, *Same* has fairly good latency, but not excellent. If optimizations could decrease the update latency, it would be well-worth it, because it allows more devices to share state concurrently, as opposed to just one shared object as in the example application.

2. The master recovery has worked well, but should be thoroughly tested. Automatic tests that disrupt the master recovery at various steps would be a good addition to this project.

3. In order to tackle different types of devices in the network, *Same* should switch to using the Bully Algorithm in order to get a master with high performance. This decreases the update latency for all devices in the network.

4. This project has not considered security issues. Any security features remains as further work.

5. Finally, the network directory server used in this project is too simple. A better network directory server could try to deal with NAT routers and have more features. Alternatively other types of discovery could be considered.

6.3 Lessons Learned

Developing a distributed, highly concurrent system was a humble experience. The development started with only simple knowledge of Java concurrency and little knowledge of the Android concurrency model. In the early stages of the project, Java's synchronization primitives were used directly. This caused

several problems: First, the locking semantics of *Same*'s classes were not precisely defined. This lead to deadlocks and code that was hard to test. Secondly, the Java memory model when using multiple threads was not well-understood.

Lastly, unintentionally sharing references between objects can have devastating effects. One example of this problem was experienced when developing master takeover. When a client started the takeover process, a reference to the client state was copied to the master. When the client received an update from the master, it rejected the update because the state object had was unintentionally shared between the client and the master and therefore already updated by the master. The system *almost* worked, but did not send notifications of change (which resulted in failure in completely unrelated components that relied on update notifications). Fortunately this bug was discovered quickly because of a failing unit test.

Studying the Java Concurrency API and proper usage of concurrency patterns was well worth the time. Blocking queues, immutable and atomic variables made concurrency much easier to handle in the later stages of the project. Some experimentation with lock-free objects was performed, with mixed results. This was discussed in Section 4.6.

Switching from synchronous to asynchronous RPC caused several problems. Asynchronous RPC suited this project better in terms of concurrency and error handling but, because some of the classes had been designed for synchronous RPC some problems arose. An example of this were the thread starvation problems mentioned in Section 4.2. Some classes contained code that was no longer optimal, such as spawning threads to handle RPCs and producer-consumer queues that were no longer needed.

6.4 Conclusion

With the increase in performance of mobile devices it is possible to adopt new development techniques in order deliver better games to this growing market.

In this project we proposed a distributed model for sharing state. The model makes it possible to create multiplayer games that use a local Wi-Fi network to communicate. With a centralized server architecture state operations are sent to a server on the internet. *Same* sends state operations between the devices using the local network only.

Several distributed shared object systems exist with different semantics and implementations. Such systems may be optimized for update latency, scalability or concurrent access. They are typically designed for distributed applications

on a much larger scale. To our knowledge, *Same* is the only attempt to create a distributed shared object system for mobile devices.

The performance of *Same* is acceptable and it has been successfully applied to create an interactive application with real-time sharing of state between devices.

Our contribution to mobile game development is the distributed system *Same*. It can be used to share state between devices without a centralized server. Updates to this state is sent to all the other connected devices in a reliable, consistent manner. When devices lose their connection, the system is able to recover and continues to function. To our knowledge, this is a new direction in mobile game development that may outperform existing centralized solutions.

Bibliography

[1] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*, volume 4 of *International computer science series*. Addison Wesley, 2005.

[2] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces - Principles, Patterns and Practice*. Addison-Wesley, 1999.

[3] David Gelernter. Generative communication in Linda. *ACM Trans Program Lang Syst*, 7(1):80--112, 1985.

[4] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 32(4):18--25, 2001.

[5] Andrew S Tanenbaum. *Modern Operating Systems*, volume 2 of *Prentice Hall international editions TS - GBV - Gemeinsamer Bibliotheksverbund*. Prentice Hall, 2001.

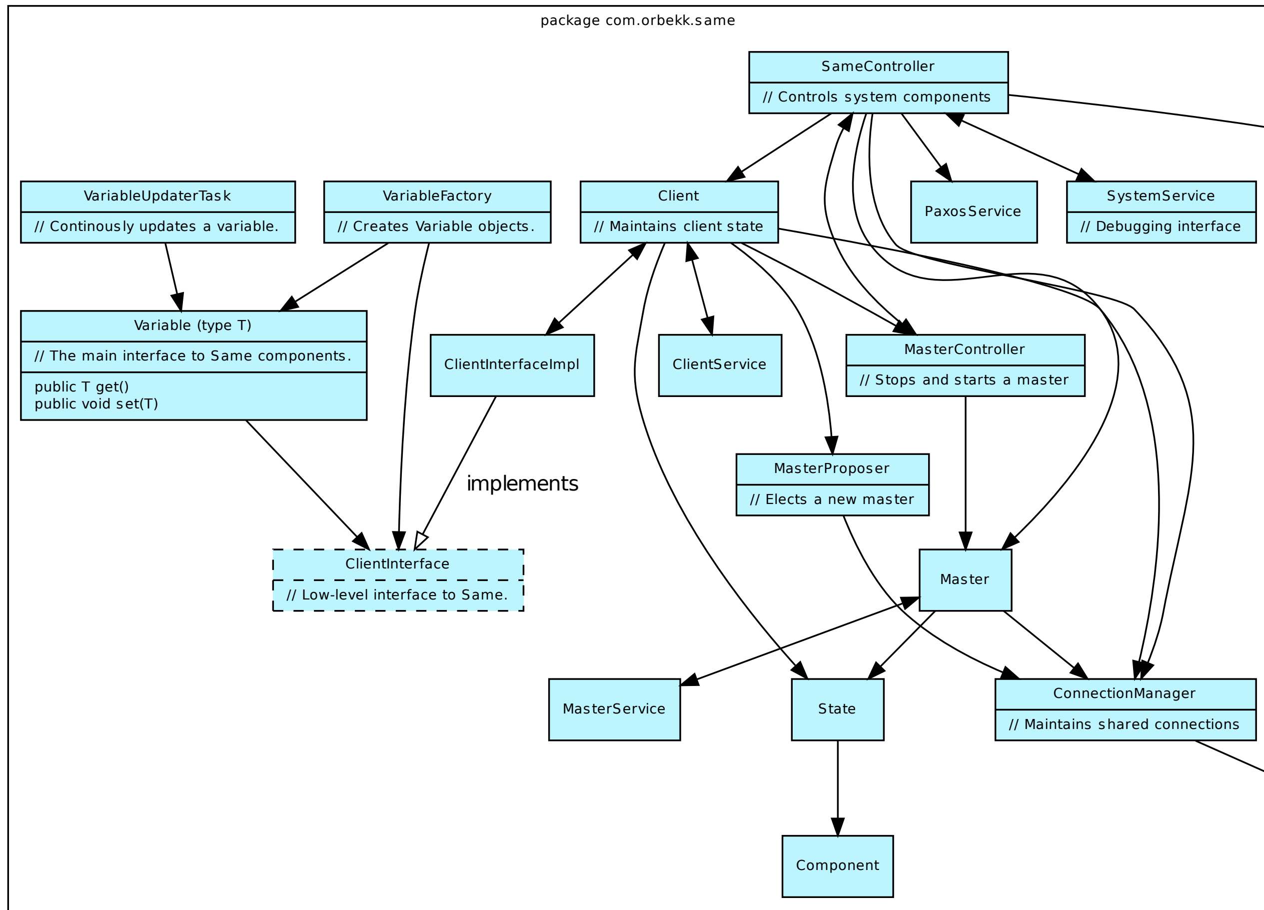
[6] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems - Principles and Paradigms*. Prentice Hall, 2002.

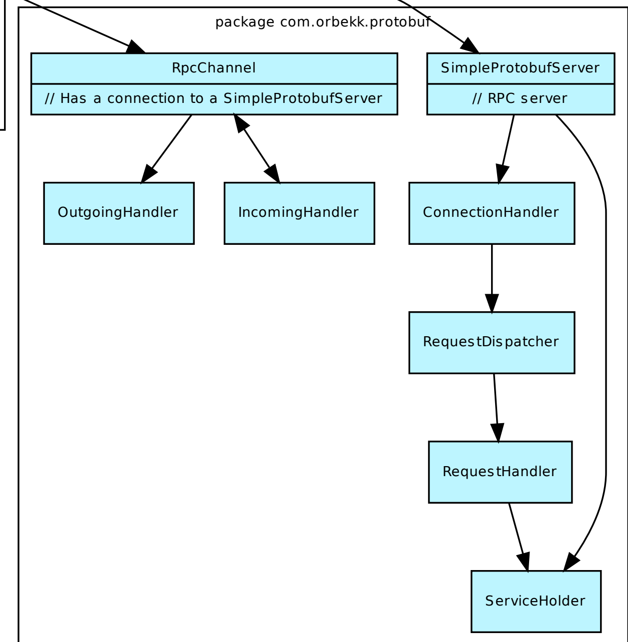
[7] Dinesh C Verma. *Legitimate Peer to Peer Network Applications: Beyond File and Music Swapping*. Wiley, 2004.

Appendix A

Class Diagram

A class diagram containing the most important classes in *Same* is shown on the next page. An arrow from class A to class B means that A interacts directly with B.


package com.orbekk.same
SameController
// Controls system components
VariableUpdaterTask
// Continously updates a variable.
VariableFactory
// Creates Variable objects.
Client
// Maintains client state
PaxosService
SystemService
// Debugging interface
Variable (type T)
// The main interface to Same components.
public T get()
public void set(T)
ClientInterfaceImpl
ClientService
MasterController
// Stops and starts a master
implements
MasterProposer
// Elects a new master
ClientInterface
// Low-level interface to Same.
Master
MasterService
State
ConnectionManager
// Maintains shared connections
Component


package com.orbekk.protobuf
RpcChannel
// Has a connection to a SimpleProtobufServer
SimpleProtobufServer
// RPC server
OutgoingHandler
IncomingHandler
ConnectionHandler
RequestDispatcher
RequestHandler
ServiceHolder

Appendix B

Test Devices Overview

Seven Android devices were used in the benchmarks covered in Section 5.2. This is an overview of the specifications of the devices. The devices are numbered 1-7. Their numbers correspond to the respective points on the graphs, e.g., the results with 3 devices were gathered using device 1-3.

Device 1 Samsung Galaxy S2

- **CPU** Dual-core 1.2 GHz Cortex-A9
- **RAM** 1 GB
- **Android version** CyanogenMod 9 Android 4.0.4
- **Provided by** Trygve André Tønnesland

Device 2 HTC Desire HD

- **CPU** Single-core 1 GHz Scorpion
- **RAM** 768 MB
- **Android version** HTC Sense Android 2.3.3
- **Provided by** NTNU Department of Computer and Information Science

Device 3 Samsung Galaxy S

- **CPU** Single-core 1 GHz Cortex-A8
- **RAM** 512 MB
- **Android version** CyanogenMod 9 Android 4.0.3
- **Provided by** Kjetil Ørbekk

Device 4 Samsung Galaxy S2

- **CPU** Dual-core 1.2 GHz Cortex-A9
- **RAM** 1 GB
- **Android version** Samsung Android 4.0.3
- **Provided by** Simen Andresen

Device 5 HTC Desire HD

- **CPU** Single-core 1 GHz Scorpion
- **RAM** 768 MB
- **Android version** HTC Sense Android 2.3.3
- **Provided by** NTNU Department of Computer and Information Science

Device 6 Samsung Nexus S

- **CPU** Single-core 1GHz Cortex-A8
- **RAM** 512 MB
- **Android version** Google Android 4.0.4

Device 7 Amazon Kindle Fire

- **CPU** Dual-core 1 GHz OMAP 4 4430
- **RAM** 512 MB
- **Android version** CyanogenMod 9 Android 4.0.4
- **Provided by** Trygve André Tønnesland