NTNU
Innovation and Creativity

Real-Time Online Multiplayer Mobile Gaming

Martin Jarrett
Eivind Sorteberg

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Alf Inge Wang, IDI
Co-supervisor: Anne Marthe Hjemås, Telenor

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

Problem Description

The main goal of this project is analyzing to what degree mobile phones can be used to play real-time online multiplayer games using today's existing mobile network technologies. When performing this analysis, measuring performance, response time, and playability will be emphasized.

In our depth study "Proximity Based Multiplayer Games For Mobile Phones", we developed a framework for developing Peer-to-Peer games for mobile phones, as well as a prototype game. This master thesis will use the framework and prototype game as a basis, but adapt them to a client-server based architecture.

Different network technologies will be tested and measured to explore which are, and which are not, suitable for this kind of games. The technologies that will be evaluated are GPRS, EDGE, UMTS (3G), and WLAN. In addition, different transfer protocols will be tested and evaluated. The most interesting aspects of these measurements will be the response time and transfer speed.

While further developing the game framework and prototype game, encountered challenges and problems will be recorded, so that the experiences gained through this project may be utilized by future developers and service providers working with similar projects.

For the testing, test modules will be developed. These modules will be flexible enough to use both on the server- and the client side application, so that relevant data can be measured both correctly and sufficiently.

Assignment given: 22. January 2007
Supervisor: Alf Inge Wang, IDI

Abstract

Gaming on mobile phones is a business with a great growth potential both in profit and popularity. In today's modern world, the number of potential users of online multiplayer mobile games is enormous. This is because of the wide deployment of mobile phones and the increasing general interest in gaming. For game developers, this is an interesting business area, since mobile games are faster and easier to develop than console or computer games, due to the mobile games' smaller size and reduced complexity. Telecom companies, on the other hand, may profit from this both by attracting users through exclusive contents only available to their subscribers, and trough the potential network traffic generated by online multiplayer games. Some multiplayer mobile games are available on the market today. However, few of these can be played real-time, which often involves a more entertaining and attractive gameplay compared to slower, turn-based games.

This project has focused on two main areas. Firstly, different network technologies and transport protocols have been tested to evaluate whether these are suitable for real-time multiplayer mobile games or not. This was done by testing the different networks' response times and transfer speeds. Secondly, a framework for developing this kind of games has been developed. Also, a game prototype has been implemented based on this framework, and the experience from this development has been recorded to provide assistance for future development projects within the same scope.

The results from the tests show that, among the widely available mobile networks today, only UMTS (3G) and EDGE offer performance sufficient for a fast and stable real-time multiplayer mobile game. GPRS is too slow and unstable, and using this technology for real-time game communication is likely to lead to lags and an incoherent gameplay. Furthermore, the tests have clearly shown that UDP is far better suited for in-game communication than TCP, because of UDP's superior response time.

For developers of such games, there are several challenges that have to be closely considered. Synchronization of clients is a very difficult task because of high network latencies. Furthermore, mobile phones are weak in terms of available resources. Managing these problems requires distribution of calculations and efficient algorithms. The game framework developed in this project has proved to provide a good basis for developing different game concepts within real-time multiplayer mobile gaming. Common functionality for such games is implemented in the framework, thus helping game developers avoid having to reinvent the wheel.

This project has shown that successful real-time multiplayer mobile games are definitely possible to implement. However, doing this is a great challenge, both for developers, distributors, and telecom companies offering such games to their subscribers. A middle way has to be found between the complexity of the game, the need for frequent network updates, and the user cost involved with playing the game. If this middle way is found, it is very likely that such a game could be a great success.

Preface

This project was performed as a master thesis in *TDT4900 Computer Science*, the conclusion of the Master of Science degree in Computer Science at the Norwegian University of Science and Technology (NTNU). The participants in this project were Martin Jarrett and Eivind Sorteberg, and the work was carried out from January to June 2007. The project description was outlined by the the project participants, the project supervisor Alf Inge Wang at Department of Computer and Information Science (IDI) at NTNU, and Anne Marte Hjemås at Telenor.

Acknowledgments

First, we want to thank Alf Inge Wang for his vital and conclusive help, support, guidance, and advices as the project's supervisor during the project work.

We also want to express our gratitude to Anne Marte Hjemås and Telenor for supporting this project with ideas and feedback as well as test mobile phones and subscriptions.

Finally, we want to thank our fellow master candidates Ole Kristian Mørch-Storstein and Terje Øfsdahl for exchange of ideas, for performing valuable testing, and for giving important feedback.

Trondheim, June 22, 2007

Martin Jarrett Eivind Sorteberg

Contents

I Introduction 1

1 Introduction 3
1.1 Motivation . . . 3
1.2 Problem Definition . . . 4
1.3 Project Context . . . 4
1.4 Reader's Guide . . . 5

2 Research Questions and Methods 7
2.1 Research Questions . . . 7
2.2 Research Methods . . . 8
2.2.1 Research Approaches . . . 8
2.2.2 Technology Validation Methods . . . 9

3 Development Methods and Tools 11
3.1 Development Methods . . . 11
3.1.1 eXtreme Programming . . . 11
3.1.2 Unified Process . . . 13
3.1.3 Our Development Method . . . 14
3.2 Development Tools . . . 17

II Prestudy 19

4 Central Concepts 21
4.1 Client-Server Networking . . . 21
4.1.1 Client types . . . 22
4.2 Mobile Gaming . . . 22
4.2.1 Multiplayer Mobile Gaming . . . 23
4.2.2 Client-Server Multiplayer Mobile Gaming . . . 24
4.3 Framework . . . 24

5 Technology 27
5.1 Java Platform, Micro Edition . . . 27
5.1.1 Java ME Architecture . . . 27
5.1.2 Limitations . . . 30
5.2 Mobile Phones . . . 30
5.2.1 Features . . . 30
5.2.2 Requirements for Mobile Phones in this Project . . . 32
5.2.3 Our Test Phones . . . 32

5.2.4 Emulators . . . 33
5.3 Mobile Network Technologies . . . 33
5.3.1 2G Networks . . . 33
5.3.2 3G Networks . . . 35
5.3.3 WLAN/WiFi . . . 37
5.3.4 Comparison . . . 38
5.3.5 Other Networks . . . 38
5.4 Transport Protocols . . . 39
5.4.1 TCP . . . 40
5.4.2 UDP . . . 40
5.4.3 SCTP . . . 41
5.4.4 Comparison . . . 41

6 State-of-the-Art **43**
6.1 Today's Multiplayer Mobile Games . . . 43
6.1.1 Pirates of the Caribbean Multiplayer Mobile . . . 43
6.1.2 Samurai Romanesque . . . 45
6.1.3 Tibia Micro Edition . . . 46
6.1.4 Summary . . . 48

III Own Contribution **49**

7 Prototype Game **51**
7.1 BrickBlock . . . 51
7.1.1 Game Concept . . . 51
7.1.2 Game Rules . . . 52
7.1.3 Game Objects . . . 52
7.2 Game Framework . . . 53

8 Server- vs. Client-side Calculations **55**
8.1 Collisions . . . 56
8.1.1 Collision Detection . . . 56
8.2 Game Control . . . 62
8.2.1 Power Ups . . . 62
8.2.2 Game Settings . . . 63
8.2.3 Object Positioning . . . 63
8.3 Summary . . . 64

9 Game Flow **67**
9.1 Movement Prediction . . . 67
9.2 Masking Delay . . . 69
9.3 Message Bundles . . . 71
9.4 Summary . . . 72

10 Requirements **73**
10.1 Client . . . 73
10.1.1 Functional Requirements . . . 74
10.1.2 Non-functional Requirements . . . 77
10.2 Server . . . 79
10.2.1 Functional Requirements . . . 79
10.2.2 Non-functional Requirements . . . 81

11 Architecture **85**
11.1 Classes . . . 89
11.2 Communication . . . 90
11.2.1 Protocols . . . 91
11.2.2 Message Format . . . 92
11.3 Models . . . 96
11.3.1 Client Models . . . 96
11.3.2 Server Models . . . 97
11.4 Views . . . 99
11.4.1 Client Views . . . 99
11.4.2 Server Views . . . 100
11.5 Threads . . . 100

12 Test Modules **103**
12.1 Response Time . . . 103
12.2 Transfer Speed . . . 104

IV Test Results and Evaluation **107**

13 Test Results **109**
13.1 Formulas . . . 109
13.2 Response Time . . . 110
13.3 Transfer Speed . . . 113
13.4 Game Data Transfer . . . 115
13.5 Large Data Packets . . . 117
13.6 Summary . . . 119

14 Problems Encountered **121**
14.1 Java Related Problems . . . 121
14.1.1 The `Connection` Classes . . . 121
14.1.2 Using language level 5.0 with JWT 2.5 . . . 122
14.1.3 Heap Address Error . . . 123
14.2 Algorithmic Problems . . . 124
14.2.1 Movement Prediction . . . 124
14.2.2 Pushing Other Players . . . 126
14.2.3 Difference in Player Speeds . . . 128
14.3 Other Problems . . . 129
14.3.1 Changed Test Results . . . 129
14.3.2 UDP Response . . . 130
14.3.3 Randomly Generated UDP Packets . . . 130

15 Fulfillment of Requirements **131**
15.1 Client Requirements . . . 131
15.1.1 Functional Requirements . . . 131
15.1.2 Non-functional Requirements . . . 134
15.2 Server Requirements . . . 135
15.2.1 Functional Requirements . . . 135
15.2.2 Non-functional Requirements . . . 137

16 Method Evaluation **141**
16.1 Research Methods . . . 141

16.2 Development Methods 142

17 Technology Evaluation 147
17.1 Mobile Network Technologies 147
17.1.1 GPRS 147
17.1.2 EDGE 148
17.1.3 UMTS 148
17.1.4 WLAN 148
17.1.5 Conclusion 148
17.2 Transport Protocols 148
17.2.1 TCP 149
17.2.2 UDP 149
17.2.3 Conclusion 149

V Summary 151

18 Answers to Research Questions 153
18.1 Multiplayer Mobile Game's Challenges 153
18.1.1 Developer Challenges 153
18.1.2 Service Provider Challenges 154
18.2 Mobile Network Technology 155
18.2.1 Response Time 155
18.2.2 Transfer Speed 155
18.2.3 Playability 156
18.2.4 Cost 156
18.3 Gameplay 157
18.3.1 Synchronization 157
18.3.2 Connection Management 157
18.4 Multiplayer Mobile Game Framework 158

19 Conclusion 161

20 Further Work 163
20.1 Extending BrickBlock 163
20.1.1 Adding Bots 163
20.1.2 Gameplay and Game Content 164
20.1.3 Improving the Force Push Algorithm 164
20.2 Extending the Framework 165
20.2.1 Smooth Turning 165
20.2.2 Confirmation of Critical Data Receival 165
20.2.3 Other Networks and Protocols 166
20.3 Further Testing 166
20.3.1 Stability 166
20.3.2 Usability Testing 166

21 Recommended Readings 169
21.1 Game Development with Java ME 169
21.2 Game Development in General 170
21.3 Mobile Networks and Transport Protocols 170

Bibliography **170**

VI Appendices **175**

A Glossary **177**

B Running BrickBlock **181**
B.1 Running the BrickBlock Client . . . 181
B.2 Running the BrickBlock server . . . 182

C Extended Backus-Naur Form **185**

D Detailed Architecture **187**
D.1 BrickBlock Messages . . . 187
D.1.1 Message Specifications . . . 187
D.1.2 Example Messages . . . 188
D.2 Sequence Diagrams . . . 189
D.2.1 Joining an Active Session . . . 189
D.2.2 Interacting in the Lobby . . . 190
D.2.3 Playing a Game . . . 191
D.2.4 Administrating a Game . . . 193
D.2.5 Detecting Disconnections . . . 193
D.3 Extending the Framework . . . 195
D.3.1 Client . . . 195
D.3.2 Server . . . 197
D.4 Class Diagrams . . . 199

E Files **201**
E.1 Applications . . . 201
E.1.1 Client . . . 201
E.1.2 Server . . . 201
E.2 Class Diagrams . . . 201
E.2.1 Client . . . 201
E.2.2 Server . . . 201
E.3 Javadoc . . . 202
E.4 Source Code . . . 202
E.4.1 Client . . . 202
E.4.2 Server . . . 202
E.5 Test Results . . . 202
E.5.1 Response Time . . . 202
E.5.2 Transfer Speed . . . 202
E.5.3 Large Data Amounts . . . 203

List of Tables

2.1 Technology validation methods 9

5.1 Available test phones 32
5.2 Current 802.11x WLAN standards 37
5.3 Comparison of mobile network technologies 38
5.4 Proposed 802.11n specifications 39
5.5 Transport protocols 41

9.1 Warp distances 69

10.1 Functional requirements for the client application 77
10.2 Functional requirements for the server application 82

11.1 List of actions used in the game framework. 95
11.2 List of actions specific for BrickBlock. 96

13.1 Statistical values of the response time test results with send interval 100 ms and 250 ms 113
13.2 Statistical values of the transfer speed test results with packet size of 120 bits and 360 bits 114
13.3 The setup on the test phone 115
13.4 Data amounts and cost with TCP 116
13.5 Data amounts and cost with UDP 116
13.6 Data amounts and cost with UDP while playing the game 117

B.1 Valid arguments when starting a server 183

C.1 EBNF notation 185

List of Figures

3.1 Work distribution in UP 14

4.1 Conceptual model of a client-server network 21

5.1 Java ME architecture 28

6.1 Screenshot of Pirates of Caribbean Multiplayer Mobile Game (2006) from Floodgate Entertainment 44
6.2 Screenshot of Samurai Romanesque (2001) from Dwango 46
6.3 Screenshot of Tibia Micro Edition (2006) from CipSoft 47

7.1 Conceptual model of the BrickBlock game 52
7.2 The game objects in BrickBlock 53

8.1 Self-caused wall collision 56
8.2 Externally caused wall collision 57
8.3 Player collisions with simultaneous movement 59
8.4 Server-side collision handling 60
8.5 Client-side collision handling 61

9.1 Movement prediction 68
9.2 Interpolating smooth turning 70
9.3 The two different methods for sending messages 72

10.1 Client state chart 74
10.2 Pre-game screenshots 75
10.3 Server state chart 79

11.1 Architectural overview 85
11.2 Client architecture 87
11.3 Server architecture 88
11.4 High-level client class diagram 89
11.5 High-level server class diagram 90
11.6 Format of the string messages sent from client to server 94
11.7 Format of the string messages sent from server to client 94
11.8 Client model representation 96
11.9 Server model representation 98
11.10 Server threads 100
11.11 Communication threads 101

12.1 Response time test 104
12.2 Transfer speed test 105

13.1 Measured response time 111
13.2 Measured response time including transmission interval 112
13.3 Measured transfer time 114
13.4 Measured transfer time with large data packets 118
13.5 Locating the maximum packet size 119

14.1 Illustration of the missed stop packet problem 126
14.2 Illustration of the force push problem 127
14.3 Player movement polling 128

D.1 Joining an active session 190
D.2 Interacting in the lobby 191
D.3 Client game sequence 192
D.4 Server game sequence 194
D.5 Alive requests 195

Listings

8.1 Procedure for handling player collisions . . . 61
9.1 Procedure for predicting movement . . . 68
9.2 Formula for prediction error . . . 68
11.1 The `Communicator` interface . . . 91
11.2 The server's `MessageParser` interface . . . 92
11.3 The client's `MessageParser` interface . . . 92
11.4 EBNF representation of the message format . . . 93
14.1 Use of `instanceof` on `Connection` objects . . . 122
14.2 Using a `StringBuffer` for building strings . . . 123
14.3 Boxing primitives to avoid compiler error . . . 123
14.4 Using movement speed per time unit . . . 129
B.1 Running server with default values . . . 182
B.2 Running server with specified values . . . 183
B.3 Running test with default port number . . . 183
B.4 Running test with specified port number . . . 183
C.1 An example of grammar rules defined with EBNF . . . 185
D.1 Client ⇒ server message specification . . . 187
D.2 Server ⇒ client message specification . . . 188
D.3 Examples of messages sent from client to server . . . 188
D.4 Examples of messages sent from server to client . . . 189

Part I

Introduction

Chapter 1

Introduction

This chapter contains a section on the motivation behind this project. It also includes a section with an explanation of the project's problem definition and a section about the project context. At the end of the chapter, we have included a reader's guide to aid readers to quickly find the parts relevant for various readers.

1.1 Motivation

It has become more and more usual to download content to one's mobile phone. Examples of such content are ringtones, music, graphics, or singleplayer games. Such content are used to personalize the mobile phone and for entertainment. The content providers make money on the sale or on subscription to updates, while the network operators/carriers make money on the data downloaded. This type of content can also be shared between mobile phones using free network technology like Bluetooth.

For the most part, mobile phone games are small games with simple gameplay and a limited timespan. This means that they are easy to start, and that a gaming session can be started and finished within a few minutes. The player can play while waiting for a bus, being bored in a lecture, or in similar scenarios. The display and the user input interface on mobile phones also greatly dictate and limit the game design. Multiplayer games are more engaging than singleplayer games because in multiplayer games, the player competes with real people instead of computer opponent. This results in more involvement from the player and more addiction to the game compared to a singleplayer game. However, multiplayer games demand an entertaining gameplay and a satisfactory player experience. These qualities will lead to players playing the game more.

Multiplayer mobile phone games yield high demands to performance from the mobile phone and the network technology. Development issues to consider are synchronization between players, size of data to be exchanged, how often data needs to be exchanged, generation and parsing of data, data overhead, data transfer speed, and handling data losses. These issues have different impacts on the gameplay dependent of the game type. For example, a real-time action game has a higher demand for synchronization and frequent data updates than a turn-based strategy game has.

Mobile phone carriers are interested in using mobile games to provide an exciting service to

their users, attract new users, and profit from the users playing games. If a game is very good, entertaining, and "cool", the game can become popular. The better the game is, the more likely it is to become popular. The game's popularity can create a rub-off effect leading to people wanting to play the game, which can be used for attracting new subscribers to a carrier. This is especially true if the game is only available through one carrier. "Killer apps", or exclusive, good games have been used to promote and sell game consoles, and is an effective way to promote a specific carrier, subscription, or mobile phone. Carriers can profit from multiplayer mobile games in many ways besides only the data traffic generated by the games. A game can be subscription based, which means that a player must pay a fee for playing the game. Through services such as hosting game downloads and game servers, a carrier can collaborate with other parties responsible for developing and maintaining mobile games. The parties can then share the revenues generated by the game.

Java ME is the most used programming platform for applications for mobile phones. Today, almost every mobile phone manufacturer delivers phones with support for Java ME [50]. Those that do not, are either too small to be considered to have an impact on the mobile phone market, or they are targeting their mobile phones for specific use where Java ME support is not needed. With Java ME, applications and games can easily be developed by software developers.

1.2 Problem Definition

The main goal of this project is analyzing to what degree mobile phones can be used to play real-time online multiplayer games using today's existing mobile network technologies. When performing this analysis, measuring performance, response time, and playability will be emphasized.

In our depth study "Proximity Based Multiplayer Games For Mobile Phones" [29], we developed a framework for developing Peer-to-Peer games for mobile phones, as well as a prototype game. This master thesis will use the framework and prototype game as a basis, but adapt them to a client-server based architecture.

Different network technologies will be tested and measured to explore which are, and which are not, suitable for this kind of games. The technologies that will be evaluated are GPRS, EDGE, UMTS (3G), and WLAN. In addition, different transfer protocols will be tested and evaluated. The most interesting aspects of these measurements will be the response time and transfer speed.

While further developing the game framework and prototype game, encountered challenges and problems will be recorded, so that the experiences gained through this project may be utilized by future developers and service providers working with similar projects.

For the testing, test modules will be developed. These modules will be flexible enough to use both on the server- and the client side application, so that relevant data can be measured both correctly and sufficiently.

1.3 Project Context

This project is a master thesis carried out as a part of NTNU's research program on video games, which is organized under IDI. Parts of this master thesis is based on results from the

depth study "Proximity Based Multiplayer Games For Mobile Phones" [29] completed during the fall semester 2006 at IDI at NTNU.

The supervisor for this master thesis is a member of the Software Engineering Group at NTNU, while the participants are Master of Science students at the same group. The research profile of the group covers the field of software quality and software process improvement (SPI), as well as process modeling/enactment, software architecture, configuration management, object oriented programming and reuse, and distributed systems.

This project is also supported by, and performed on behalf of, Telenor R&I, which is the research and innovation unit of Telenor. Telenor is the largest telephone company in Norway and one of the largest mobile operators worldwide [54]. The interest areas for Telenor are the testing and utilizing of the network technology in addition to the process of developing multiplayer mobile games. They will use this information in future mobile games projects, motivated by reasons discussed in Section 1.1. Telenor R&I supports this project by providing a test environment, resource persons, and help and guidance from research scientist Anne Marte Hjemås.

1.4 Reader's Guide

Part I Introduction explains the motivation behind the project, the problem definition, the project context, and the research questions and methods used. It also lists the development method and tools used to create the project's applications and write the project report. This part is most relevant for readers interested in the reasoning behind this project as well as the project's purpose.

Part II Prestudy describes central concepts for this project, relevant technologies, and a discussion of today's top multiplayer mobile games. This part is relevant for readers interested in the functionality, features, and properties of the different technologies used in the project.

Part III Own Contribution contains a description of the prototype game and the game framework developed in this project. A discussion of important aspects such as where calculation should be performed and which methods can be implemented to ensure good game flow are performed. The part concludes with representation and explanation of the requirements, architecture, and test modules of the applications developed. This part is for readers wanting to understand the reasoning behind the development of the project applications and their architecture. Future projects aiming to develop real-time multiplayer mobile games can find many useful discussions and considerations here.

Part IV Test Results and Evaluation consists of the test results and the evaluation of this project. The formulas used to calculate the test results are presented and the results are presented, discussed, and a comparison between the mobile network technologies and transport protocols are performed. The evaluation starts with a discussion of the problems encountered in the project work and continues with a discussion of the fulfillment of the requirements. Also, the project's research and development methods are evaluated. Finally, the technologies used in the project are evaluated. This part is especially interesting for readers interested in the main result of the project and the project progress. It also is useful for readers who want to know about the benefits and drawbacks of the technology used.

Part V Summary contains the project's conclusion, possible further work for the project, and a list of recommended readings for other projects with a similar scope. Readers

interested in the main results, and the discussion around these, should also read this part. Also, it is useful for readers interested in further work within this project's context, or starting a project within a similar research field.

Part VI Appendices consists of the project's glossary, a manual for running the developed game, and an explanation of the EBNF notation. Furthermore, the appendices provide a more detailed description of the developed applications, presented by class diagrams and sequence charts. Also, a detailed description of the message formats used in the applications are included. This part is most interesting if one wants to get a deeper understanding of the implementation of the game framework and the game prototype. Also, the glossary chapter (Appendix A) is helpful for understanding the abbreviations used in the report.

Chapter 2

Research Questions and Methods

In this chapter we identify the research questions we will answer through our project, and the methods we will use to answer them.

2.1 Research Questions

In this project we consider the research questions listed in this section. The research questions are meant to specify the expected output of this project and thereby constrain the direction of the project work. These questions are created to derive answers to the aspects mentioned in the Problem Definition Section 1.2. The important outcome of these questions are the pitfalls and challenges of a real-time multiplayer mobile game project, and how mobile phones and existing mobile network technologies are suited for such games.

1. **Which challenges exist when developing real-time multiplayer client/server-games for mobile phones?**
   (a) What are the developer specific challenges?
   (b) What are the service provider specific challenges?
2. **Do existing mobile network technologies for mobile phones provide satisfying properties for real-time multiplayer mobile phone games?**
   (a) What are the differences between GPRS, EDGE, UMTS, and WLAN in terms of response time?
   (b) What are the differences between GPRS, EDGE, UMTS, and WLAN in terms of transfer speed?
   (c) What are the differences between GPRS, EDGE, UMTS, and WLAN in terms of playability?
   (d) How does the amount of data transfered affect the associated user cost when using the different network technologies?
3. **What problems need to be solved to ensure a satisfying gameplay and how can they be solved?**
   (a) What methods can be used to ensure the game is sufficiently synchronized?

(b) What methods can be used to allow dynamic connections and disconnections?

4. **What requirements need to be fulfilled when developing a client/server multiplayer game framework?**

2.2 Research Methods

Our research questions listed in the previous section concern several different areas, and can and should be approached in different ways in order to be answered satisfactory. While some of the questions are easiest answered through development and experimentation, others are better answered by performing and evaluating different tests. In this section, we will give a short introduction to the research methods we will use to answer our research questions.

2.2.1 Research Approaches

Within software engineering, Basili [4] identifies three main approaches to answer research questions where experiments can be performed and results measured.

The Engineering Method is a scientific method where engineers build and test a system based on a hypothesis. This method makes use of iterations, where results are observed and evaluated for each iteration. These results are then used to improve the system, before the cycle is repeated. The iterations are then repeated until no further improvements are needed.

The Empirical Method is the other of the research methods classified as scientific methods. Using this method, a statistical method is proposed to validate a given hypothesis. There may not be a formal model or theory describing this hypothesis. Using the proposed method, data is then collected to verify or falsify the hypothesis.

The Mathematical Method is an analytic method, where a formal theory is proposed, or derived from a set of axioms. The results from this theory can then be compared with empirical observations to verify the validity of the theory.

Even though these three methods are quite different, some or all of them can still be used within the same project to help answer different kinds of questions. As mentioned, the nature of our research questions are quite varying, and the method(s) used to answer each question should therefore be based on that particular question's nature.

Three of the four main questions listed in the previous section can be seen to have similar nature. These questions are questions 1, 3, and 4. For all of these questions, obtaining a satisfactory answer requires gaining knowledge and experience. This kind of experience can typically be gained through developing such games, and then evaluating the experience gained through this development. Possible problems can then be recorded and discussed, and suggestions to the problems can be proposed. This approach is just like that of the *Engineering method*, and this method should therefore be used to answer these questions.

For the last main question stated in the research questions, question 2, another approach is needed. While this question *can* be answered through observing the properties of each technology, the differences between the technologies may be so small that obtaining exact answers is very hard. Furthermore, the conclusions have a risk of being biased, as there is a risk of expecting that one technology is "better" than another, and base the observations on these opinions.

Thus, a better approach to answer this question is using the *Empirical method*. For each of the subquestions, objective measurements can be obtained, and these measurements can then be evaluated using statistical methods. This ensures an unbiased basis for the answers to the questions. Furthermore, the results of the measurements can be recorded and used in later analysis if needed.

2.2.2 Technology Validation Methods

The research methods mentioned above describe the methods we will use to answer our research questions. In addition to these methods, Wang [61] describes twelve methods used for technology validation, divided into three main categories. These methods are listed in Table 2.1.

Table 2.1: Technology validation methods

Observational	**Historical**	**Controlled**
Project monitoring	Literature search	Replicated experiment
Case study	Legacy data	Synthetic environment experiments
Assertion	Lessons learned	Dynamic analysis
Field study	Static analysis	Simulation

For our project, three of these methods stand out as particularly useful. The first two of these are the historical *Literature search* and *Lessons learned* methods. Both of these methods emphasize extracting experience from previously completed projects. When using *Literature search*, results of papers and other documents are analyzed to confirm an existing hypothesis, or to improve the data collected in one project with more similar data. We will use this method to find and evaluate existing games similar to the prototype we will develop. The results of these evaluations will then be used to obtain knowledge about relevant aspects with this kind of game development and deployment.

In the *Lessons learned* method, qualitative aspects of previous projects are used to improve future projects. As previously mentioned, this master thesis is based on our depth study [29]. Naturally, our experience from the depth study will have a major influence on this project, and we will try to utilize that experience as far as possible.

The third and final technology validation method we find to be useful is the *Simulation* method. This method uses a small model to simulate a real environment. The real environment for a multiplayer mobile game implies enormous amounts of locations and simultaneous users. The size of this project is far too small to extensively test this environment, and we will therefore use simulation to simulate typical environments, and base our conclusions on these simulations.

Chapter 3

Development Methods and Tools

In this chapter, we first describe the development methods we will use during this project. At the end of the chapter, a list of the development tools we will use is provided, along with a short description of each tool.

3.1 Development Methods

This project differs from a standard development project, in that the main goal is not actually implementing a product ready for use, but rather running tests and evaluating the suitability of existing network technologies for multiplayer mobile gaming.

Because of this, the high-level requirements can be, and should be, defined early in the project period, and we can expect them to be pretty stable during the entire project. However, we do need to use a development method that allows for detecting and handling risks at early stages. We need to develop test modules and communication interfaces within areas that none of us have much previous experience. This may easily lead to unpredicted problems that require more time than expected. Because of this, a flexible development method that allows for changing work schedules is needed. Another frequently used term for this kind of development methods are agile methods, and two examples of such methods are the eXtreme Programming and Unified Process methods [34].

In this section, we describe these two development methods, and their key practices. At the end of the section, we extract the key points from these two development methods and use these to form our own development method, custom-made for this particular project. The final part of this section will then present this development method, and the benefits we hope to achieve through using this particular method.

3.1.1 eXtreme Programming

eXtreme Programming (XP) is recognized in [34] as "*a well-known agile method, that emphasizes collaboration, quick and early software creation, and skillful development practices*". When using XP, very little effort is put into planning the entire project. Instead, the method focuses on planning small, incremental tasks. With this approach, the XP method is very flexible, as changes in the project's premises can be discovered and, if not handled immediately, simply be

scheduled for a later iteration. This enables the XP method to handle changing circumstances in a far more flexible way than stricter development methods, such as the Waterfall model [46].

XP is founded on four values: *communication*, *simplicity*, *feedback*, and *courage*. These values are used to determine whether the work done using XP is done right according to the XP principles.

XP Practices

eXtreme Programming consist of a number of practices. Some of these practices are not unique for XP in themselves, but put together in an XP environment, they provide an effective framework for producing small to medium sized software projects [46]. The following list provide 14 such key practices.

Whole team, or onsite costumers The whole team, both programmers and costumers, work together in a common project room. One or more costumers sit more or less full time with the team. They are expected to be subject matter experts, and are empowered to make decisions regarding requirements and their priority.

Small, frequent releases Evolutionary delivery.

Testing: acceptance testing and customer tests All features must have automated acceptance tests. All tests must run with a binary pass or fail result, so that no human inspection of the code is required. The acceptance tests are written in collaboration with the costumer.

Testing: test-driven development and unit testing Unit tests are written for most code, and the practice of test-driven development is followed. This means that the tests should be written before the code to be tested.

Release planning game The goal of this practice is to define the scope of the next operational release, with maximum value to the software. The game is performed by the costumer writing story cards to describe features, and the developers estimating them. The next release is then planned by either setting a release date and adding a suitable amount of features, or adding features and calculating the release date.

Iteration planning game The goal of this practice is to choose the stories to implement, and plan and allocate tasks for the iteration. The costumer chooses a story card to implement, and for each, the programmers create a task list the fulfill the stories. Then the tasks are chosen through volunteering, and their lengths are estimated. If any tasks are estimated too long (half-day to two-day range), they are refactored.

Simple design Avoid speculative design for possible future changes. Avoid creating generalized components that are not immediately required. The design should avoid duplicate code, have a relatively minimal set of classes, and be easily comprehensible.

Pair programming All production code is created by two programmers at one computer, where they rotate using the input devices periodically.

Frequent refactoring This practice is also known as "continuous design improvement", and has a goal of minimal, simple and comprehensible code. This is achieved by small change steps, verifying tests, and using refactoring tools.

Team code ownership All programmers can change any code at any time. As a consequence, the code is "our", not "his" or "her" code. The result of this is that improvement of the

code can be performed at once, and the bottleneck of change requests in individual code ownership is removed. Even though modifying code one self has not written can be risky, some of this risk is removed through the other XP practices.

Continuous integration All checked-in code is continuously re-integrated and tested on a separate build machine, in an automated 24/7 process loop of compiling, running all unit tests and all or most acceptance tests.

Sustainable pace XP promotes "no overtime", as frequent overtime is often a sign of deeper problems, and does not lead to happy, creative developers, healthy families, or quality, maintainable code.

Coding standards With collective code ownership, frequent refactoring, and regular swapping of pair programming partners, everyone needs to follow the same coding style.

System metaphors To aid design communication, capture the overall system or each subsystem with memorable metaphors to describe the key architectural themes.

3.1.2 Unified Process

The other development method we find worth mentioning is the Unified Process development method, on which the more well-known Rational Unified Process (RUP) development method is based. Even though Unified Process (UP) is more strict than XP, it is still far more flexible than development methods such as the Waterfall model. In [34], UP is simply recognized as "a popular iterative process framework". Even though this method is similar to XP in many ways, the methods differ somewhat as to the durance of planning, goals of early iterations, and identification of requirements. While XP focuses heavily on using minimal time on planning before programming, UP is more flexible on this point. For example, UP allows and supports the creation of relatively detailed specifications, assuming that an onsite costumer is not going to be present [34]. As explained in the beginning of the chapter, this is well suited with our project's setting.

Within UP development, iterations are organized in four phases. In the *inception phase*, effort is put into identifying the high-level requirements of the project. This phase is usually short, and runs for only a few days without iterations. The *elaboration phase* iterations emphasize programming the risky, core architecture, whereas the *construction phase* iterations build the remainder. Finally, the *transition phase* can be classified as a system test phase, where the purpose is verifying whether or not release candidates are ready for deployment [34]. Figure 3.1 illustrates the distribution of work in these phases. The horizontal axis shows the duration of the project, and is divided into several time-boxed iterations.

Six UP best practices

Within UP, one can choose to follow a number of best practices. Larman [34] presents six best practices that represent a minimal set to focus on when using the UP method. Even though UP contain several more practices, these six form a basis where most or all should be applied to a UP development project.

Develop in short time-boxed iterations Like XP, UP uses iterative development. The time-boxes have a recommended duration of 2-6 weeks. Thorough requirements analysis should not be performed before programming, but the requirements should be refined during the iterations.

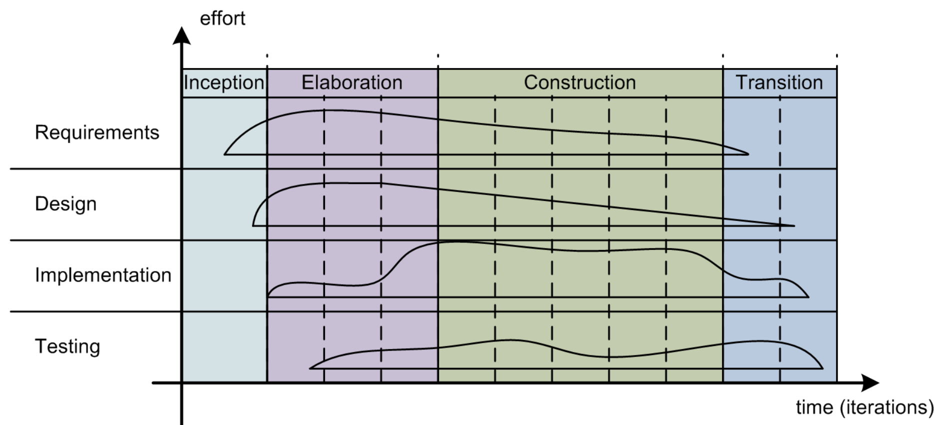

Figure 3.1: Work distribution in UP

Cohesive architecture and reuse of existing components The high-risk and high-value elements are developed first, and a cohesive architecture is emphasized in the early iterations. Reuse of existing components, large and small, reduce the amount of new code and defects.

Continuously verify quality All code is continuously tested in a realistic way by integrating and testing all the software in each iteration. This testing also extends beyond code to include early verification of usability, the quality of non-code artifacts, and of the process itself via regular team meetings.

Visual modeling At least a little modeling, such as sketching on a white board for an hour, is done before starting programming in an iteration. This helps exploring and communicating creative design ideas while ignoring low-level code details. These models will often be loosely based on the UML standard.

Manage requirements Requirements are managed through skillful means to find, organize, and track them. The requirements are found iteratively and recursively rather than through a major up-front analysis. They are organized and tracked using tools enabling the developers to see the current status of the requirements.

Manage change Change is managed using a disciplined configuration management and version control, a change request protocol, and baselined releases at the end of each iteration.

3.1.3 Our Development Method

The problem with most development methods is that they are designed for larger programming teams working on larger projects. In our case, we are only two programmers working on a project with a duration of exactly five months. Because of this, it is impossible for us to completely follow the rules of one specific development method, both because of our limited team size and the project's duration. Still, the two methods described in this chapter both contain some aspects that can definitely be worth using in our project. Because of this, we take the liberty to extract the principles we deem to be valuable for our project, and create a hybrid development method that we will use for this project. We feel that this way of adjusting the methods to meet our needs (within limits), is better than the opposite, although this may be considered controversial by supporters of one particular development method or another.

The following list contain ten practices we will follow in this project, and the outcome we expect from using these principles. For each of the practices, the development method which the practice is derived from is shown.

1. **Simple design** [XP]
   As will be further elaborated later, our main focus for this project is not creating a game with breathtaking functionality, but rather a game prototype that can be used as basis for further development. To support this, a simple design and understandable implementation will be important. Furthermore, a simple design will ease the task of identifying and improving the elements critical for a real-time multiplayer mobile game.

2. **Pair programming** [XP]
   Pair programming is a useful method for identifying and recognizing erroneous programming, as one programmer can watch and analyze the code while the other is writing. Also, this method is effective for solving difficult programming problems, as it is obvious that two heads think better than one. Still, we will not use pair programming for our entire development, as this method will be unnecessarily time-consuming for trivial tasks. But for the harder, non-trivial tasks, we will use pair programming to try to find the best solutions as quickly and efficiently as possible.

3. **Frequent refactoring** [XP]
   We can not expect to find the most suitable and effective solution at once while programming. Even though the high-level architecture of the system will be discussed and sketched before we start implementing, it is very unlikely that we will have complete control of all classes, methods, and interaction between these from the beginning. Therefore, as our system grows, it will be natural to extract functionality into appropriate classes and methods to create a logical and coherent structure. Frequent refactoring is therefore a useful method to continuously adapting the architecture to meet these goals. When using appropriate development tools, as described in Section 3.2, such refactoring is quite simple and can be performed without having to worry that the system will be broken as a result of the refactoring.

4. **Team code ownership** [XP]
   Since this project is a research project, with development within an area with little prior research, unexpected errors in our applications are very likely to occur. When such errors are discovered, it is important that they are localized and fixed as soon as possible, so that they do not affect other parts of the applications. To do this, both members of the team must be able to change the code of any part of the system at any time. This requires that both members have a good understanding of the implementation, and also that there is no distinction between "his" or "my" code. Hence, both team members must feel responsibility and ownership of all the programming code in the system.

5. **Coding standards** [XP]
   As explained when this practice was first mentioned, coding standards are important when using pair programming, team code ownership and frequent refactoring. This is true both to avoid confusion when programming, and to ensure the readability of the code for other developers looking into the code later. Like refactoring, automatic adjustment of the code can be performed by the development tools.

6. **Develop in short time-boxed iterations** [UP]
   The first of our selected UP practices ensures that the programming starts early, and that potential problems can be detected and handled early. The requirements and system design should not be decided and fixed before the development is started, but rather be refined and evolved based on the experiences of the iterations. This is one of the cor-

nerstones of all agile development methods, and also corresponds to the "*small, frequent releases*" practice of the eXtreme Programming method. Because of our short project duration, and the experimental nature of our project, we will use iterations with a duration of one week.

7. **Develop the high-risk and high-value elements first** [UP]
In any development project, some parts will always be more important than others. For example, being able to start the application is far more important than having the right colored buttons. Identifying these important requirements as early as possible, and developing them first is therefore a very effective way of ensuring a result that works satisfactory, if not perfectly. It is important to notice that this identification of key elements does not violate the practice of evolutionary requirements. Rather, it requests that the key elements are identified, so that these elements' associated requirements can be identified and evolved as soon as possible.

8. **Cohesive architecture and reuse of existing components** [UP]
In addition to identifying the critical elements as soon in the development process as possible, the Unified Process method recommends following a cohesive architecture from the beginning. This way, reusable code can be identified and extracted, and a lot of programming effort can be saved. Also, striving to use a cohesive architecture in the early iterations helps preventing the system from growing too large and complex to keep track of. The *simple design* and *frequent refactoring* practices of the eXtreme Programming method are effective tools to meet this practice.

9. **Ensure that you deliver value to your costumer** [UP]
This is not one of the mentioned six best practices for UP, but one of the several optional ones. We feel that it is very important for this project, and have decided to include it among our selected practices. In this project, we have two entities that can be considered "costumers": IDI and Telenor. Therefore, in order to deliver value to our costumers, we have to ensure that both receive satisfactory results from this project. Even though both parties, and Telenor in particular, are interested in the results of our implementation, the contents of this report is likely to be even more valuable for both parties. Therefore, we have to ensure that our development does not grow so large that the programming effort reduces the quality of our report.

10. **Manage change** [UP]
When using the iterative and evolutionary practices listed above, the properties of our project will undergo constant changes of varying degrees. To keep track of these changes, we need methods to ensure that change requests are not forgotten, and that unsuccessful changes can be undone. To address the first of these issues, we will use a list containing change requests, and whether or not these change requests have been carried out. These change requests need to describe the nature and location of the change, and its desired result. The other issue, undoing of unsuccessful changes, can be achieved through using a revision control system. For this project, we will use Subversion (SVN) for that purpose. It is important that changes are committed as often as possible to avoid conflicts when working on the same files.

As can be seen in the list, we have chosen to exclude a number of practices from both the eXtreme Programming and the Unified Process methods. Particularly, this applies to the practices involving unit testing, such as "*testing: test-driven development and unit testing*" (XP) and "*continuously verify quality*" (UP). The reason for this is that developing unit test cases is rather time-consuming, particularly for programmers that have limited experience with this method. Determining all possible situations in which the system may find itself, and

writing test cases for all of these ensures the stability of the system, but at the same time requires more time and work than we feel that we can afford for this project.

The other main category of practices we have decided to disregard are the practices involving frequent team meetings. Examples of such a practice is the "*whole team, or onsite costumers*" (XP) practice. These practices are effective for larger teams where the team members work on different parts of the project and need to meet in order to synchronize their work and plan the iterations. However, in our project, our team consists of two members, and we will be sitting right next to each other during the entire project period. Our need for set meetings is therefore non-existent. As for the practice of including the costumer in the development team, our costumer representatives (supervisors) have many other tasks than only our project. Their involvement in the project will therefore be limited to periodic meetings.

3.2 Development Tools

This chapter describes the tools and applications used in this project to develop the test applications, to write the project report, and to aid the collaboration between the project participants.

The following tools and applications have been used in this project:

IntelliJ IDEA 6.0.5 IntelliJ IDEA is a Java-based programming environment, or Integrated Development Environment (IDE), developed by JetBrains and designed to increase a programmer's productivity. It supports development in both Java EE, Java SE, and Java ME, where the last two are the most interesting aspects in our project. IntelliJ has built-in support for SVN. The support for refactoring in IntelliJ is very good, and the built-in structural search makes this kind of operations very simple and minimize the risk of breaking the code.

MiKTeX 2.5 MiKTeX is a Windows implementation of the typesetting system TeX. It includes a compiler, a LaTeX to PDF converter, and several other useful utilities [41].

TeXnicCenter Beta 7.01 TeXnicCenter is a LaTeX-editor used to easily write and structure larger documents written in LaTeX [55]. When combined with a compiler such as MiKTeX, TeXnicCenter provide a complete environment for writing and compiling LaTeX documents.

Sun Java Wireless Toolkit 2.5.1 The Java Wireless Toolkit (JWT) (formerly known as WTK) contains the packages and classes supported by the standard Java ME implementations. The toolkit is needed to compile Java source files and run these on a computer using the standard Java emulators.

Java Development Kit 6.0 A Java Development Kit (JDK) contains packages and applications needed to compile and run Java applications. The JDK 6.0 is the latest JDK from Sun and its new enhancements include improved I/O support, improved performance and security, support for generics, and an improved Virtual Machine [51].

Microsoft Office Visio Professional 2003 SP2 Microsoft Visio is a modeling tool for Windows, that will be used in this project to create diagrams and figures for the report.

Altova UModel 2007 rel.3 Altova UModel is an application for creating models in Unified Modeling Language (UML) automatically from source code. Altova UModel will be used to create class diagrams from the source code in .png file format.

Pacestar UML Diagrammer 5.08 Pacestar is a "WYSIWYG" diagramming tool that helps create UML diagrams with extensive symbol libraries and templates. This application will be used to create state charts to help modeling the functional properties of the applications developed in this project.

Part II

Prestudy

Chapter 4

Central Concepts

In this chapter we give an overview of the most vital concepts for our project. This overview covers what these concepts are, why they are important in our project, and how we plan to use them.

4.1 Client-Server Networking

The client-server architecture pattern is a network architecture pattern that separates a client with a graphical user interface from a server. Several clients can connect to the same server at the same time. Every client instance sends requests to the server and receives a corresponding response. The server usually stores and controls the data used in the application while the client displays and manipulates the data. The clients' access to data and the manipulation of it are controlled by the server, thus delivering a security control ability. The server uses a protocol to communicate with each of the client instances [5].

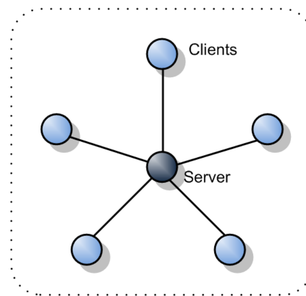

Figure 4.1: Conceptual model of a client-server network

In a client-server network, the clients do not communicate directly with each other. All communication goes through the server even for small and trivial communication. Because of this, the server may become a bottleneck in the network. Also, if the server is not working properly, the whole network goes down. Figure 4.1 shows a simplified conceptual model a client-server network [42]. An alternative to the client-server architecture is the Peer-to-Peer (P2P) architecture, where each node in the network has both client and server functionality at the same

time. All the nodes in a P2P network are responsible for contributing resources for computing, storing, and communicating data, while communication goes from node to node.

4.1.1 Client types

Clients in a client-server network can be classified as three different types. The difference between these types is the amount of local storage they need and the amount of processing they perform.

Thick clients is a client form where most of the data processing operations are performed on the client side. Because of this, local storage is also needed. With both local storage and local processing, thick clients rely very little on the server besides communication between clients. Thus, the server requirements are low, which results in cheaper and more flexible servers. Also, support for multimedia features are possible because of the high performance delivered compared to thin clients.

Thin clients implicate a minimal form of client. They use the server's resources to store and process data while their task is to display the data and possible user choices. A thin client sends requests to the server for the data needed and displays it for the user when the data is received. The users' actions are sent to the server and the server process and stores the data accordingly. Thin clients are easy to manage and have high flexibility due to the easy tasks they perform. Storage and processing operations are handled on the server.

Hybrid clients are a combination of the above mentioned types. A hybrid client can perform local processing, but will use the server for data storage. This type of client offers features from both the other types, for instance like the high performance of the thick client and the high flexibility of the thin client.

In this project we will use the hybrid client type. With an amount of local processing on the client, the gameplay will be faster and more fluid. It will also decrease the amount of network strain since it will limit the communication between the server and the client. Mobile phones have a very limited size of storage, so local storage on the client could possibly require too much. This means that the client will only have the needed data for the processing tasks.

4.2 Mobile Gaming

This section is based on our depth study [29], with the exception of Section 4.2.2, and are rewritten to fit this project's scope.

Mobile games are played on devices like mobile phones, smartphones, handheld computers, or PDAs. These types of devices are very popular and most people in the western world (i.e. "rich" countries in Europe, Northern America, and Asia) carry these devices around everywhere. This is the most positive aspect of mobile gaming, since the games can be played anywhere and anytime [13]. Mobile games are played in short breaks from everyday life to provide relaxation and a small escape from routines [43] or just to kill time [7].

The market for mobile games is related to two other business areas; the mobile telecommunications content business area and the computer game publishing business area. The mobile game business area is located between these two. Thus, this area shares many properties with the two others [43]. Key actors in this business are companies that develop games, companies

that publish games, telecommunication operators, and online mobile game portals. According to Chau [7] and Belcher [6], the worldwide market for mobile games has grown over the past years and will continue to grow because of increase in popularity, audience (the number of mobile phones with Java ME in the world and a change in the demographic), and the games' capacity (more "made for mobile phones" games). Improved distribution of mobile games, in terms of the games' availability, and lowered prices will also help selling the games. This also goes for business deals between game developers and mobile phone manufacturers to sell phones with pre-installed games. The revenues are expected grow from 3 billion US dollars in 2006 to between 7 to 17.5 billion US dollars by 2011 [7, 14, 35, 6].

For mobile games to be fun and have high quality, some qualities are especially important to strive to achieve [13]. The games need to be intuitive so the user does not have to read a lot of instructions on the small screen. This means that the game can not be too complicated and the learning curve can not be too steep. A new player must understand the game and master the central aspects of it almost immediately. The graphics should be as large as possible so that the users can play the game without holding the device close to the eyes, i.e. allow a comfortable playing position. Also, large and clear graphics enhance the understandability of the game and will compensate for the the small screen size. Another matter to consider for making the game easy to understand is to make sure that the gameplay has high simplicity. Using as few keys as possible helps the simplicity of the game, and is also smart considering the small and constricted keypad on mobile phones. Usually, mobile phones are used for gaming when waiting for a meeting, public transport, or other situations where the player has a limited free time on his hand. Because of this, mobile games should have quick and short game sessions/periods. The game should be quick and easy to start up, configure, and complete in a small amount of time, i.e. have high transience. A gaming session is limited by battery capacity as well as the player's location and situation (sitting, standing, on the move, or waiting). Because of these factors, a normal gaming session should be between a few minutes and half an hour long.

4.2.1 Multiplayer Mobile Gaming

Many of todays' multiplayer mobile games consist of uploading scores and statistics to servers to compare with other players' statistics. This concept was the first successful multiplayer mobile game type, often with limited chat features. This further extended the community feeling [45]. Other concepts make use of normal telecommunication technologies, also know as Over-The-Air (OTA) communication, to send data packets between the devices. Also, mobile games exist that use Short Message Service (SMS) or Wireless Application Protocol (WAP) for communication between players or from player to server and vice versa. These types of multiplayer games require a server and the communication between players is slow and limited. The interaction between the players is also bothersome and unsatisfactory [30].

In *Issues related to Development of Wireless Peer-to-Peer Games in J2ME* by Alf Inge Wang et al. [62], an article about issues to consider in developing wireless peer-to-peer games, the authors describe two dimensions for grouping peer-to-peer games. Even though we will use a client-server architecture instead of P2P, the game dimensions described in the article still applies to this project. This is true because the interaction between players is independent of the architecture of the system.

The first dimension discussed in the article is divided into 4 categories that describe how players interact and how the devices interact on behalf of the player. The first category is **"Controlled"**, in which players interact in specific patterns or sequences predefined by the game. In the second category, **"User interaction"**, player interaction is action triggered

by the players choice. The third category describes games that search for other players and trigger an action if one is found. This category is called **"Automatic triggered"**. The fourth category is **"Automatic"**. Games in this category interact without the players interacting with the game. The second dimension focuses on synchronization and data update between peers, and consists of 3 categories. The **"Asynchronous"** category includes games that do not need frequent update of data between peers, but can update whenever possible. In the second category, **"Synchronous"**, participating peers are dependent of frequent update of data to play the game. The last category, **"Real time"**, scopes games that need heavy data updates between players in the gameplay. Attributes like position, state, and movement are examples of information that typically needs to be sent between peers in real time.

To help with the transience aspect of mobile phone games, multiplayer mobile games need to connect to a server or opponents fast and easily. The discovery time between devices should be kept at a minimum so the connection between devices completes quickly and smoothly. For many games, the transfer speed between the devices, or to and from the server, needs to be high. The are some games that do not need high transfer speed because of small amounts of data sent between the devices or because of rare data update, but having high transfer speed is never a disadvantage. Network multiplayer games also add extra resource consumption and developers have to take into account what the network technologies demands of CPU, memory, and power. These demands must not intervene with demands from other parts of the game or the device itself.

4.2.2 Client-Server Multiplayer Mobile Gaming

With a client-server architecture in a multiplayer mobile game, the resource demands on the devices can be decreased and the game stability and availability is increased compared to other multiplayer game types such as P2P-games. A dedicated server is needed to ensure higher bandwidth, increased processing, and to provide an always-on service. The dedicated server does the most difficult and resource demanding calculations and sends the results to the clients. The server could also be divided into a database server and a game server, where the game server controls the calculations and communications while the database server controls data storage and manipulation [2]. In a Massively Multiplayer Mobile Game (MMMG) a database server would be essential for storing the vast amount of data needed for such a game. However, in this project a database server is not needed because of the type of game the prototype will be.

Because of the data transfer needed for a multiplayer mobile game, the network strain is an issue for mobile operators. They do not want the game's data transmission to occupy so much resources that the main telecommunications are effected. In addition, a multiplayer game demands low latency and delay to be satisfying to play. This means that data transmission must be highly effective and the amount of data to transfer should be kept low [45]. The bandwidth a multiplayer mobile game consumes is in direct proportion with the number of players.

4.3 Framework

By developing a multiplayer mobile game framework in this project, a basis for future projects with a similar scope is created. Such a framework will save both time and effort when developing new games since the basic functionality is already completed. Only the game specific

functionality and content has to be developed. This section is based on our depth study [29]. A discussion of which of the two framework types to develop in this project is added.

A framework in software development terms is a support structure that another software project can use in its development. According to Wikipedia [9] frameworks can be divided into two separate fields: software frameworks and application frameworks.

Software Framework A software framework is a reusable object-oriented design for a software system and consists of a set of abstract classes that collaborate for a specific type of software. The framework will help limit the choices during development, so it increases productivity, especially in big and complex systems.

Application Framework An application framework refers to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. The framework bundles a large amount of reusable code to save time in further development. By using object-oriented programming techniques to implement the framework, the unique parts of an application can inherit from the preexisting classes in the framework.

The framework developed in this project is meant to be used in multiplayer mobile game development projects with a client-server architecture. Future projects can reuse the existing classes since they contain the necessary functionality for mobile games that needs communication between clients and a server, player movement, collision detection, and player score. The functionality and game rules in the framework should be easy to extend or change. Because of this, the framework developed in this project will be an application framework.

Chapter 5

Technology

During a research project such as ours, there are a few technologies that have to be considered. Every technology in use provides some important and necessary features that are needed to complete the project. However, these technologies usually also have some weaknesses that needs to be evaluated and taken into consideration when planning the project.

This chapter lists the different technologies that will be used throughout our projects, and describes each technology with extra focus on the limitations that will influence our possibilities.

5.1 Java Platform, Micro Edition

This section is based on our depth study [29] and has been modified to address the aspects important for this project.

Java Platform, Micro Edition (Java ME), formerly known as J2ME, is a set of Application Programming Interfaces (API) targeting appliances like PDAs and mobile phones, developed by Sun Microsystems. It is optimized for appliances that are wirelessly networked and that have a relatively small amount of memory. A normal Java ME enabled device will only implement a small subset of all the available APIs. As a consequence of this, a game implemented for one particular mobile phone (or set of mobile phones) can not be expected to run on all mobile phones, even though these support Java ME.

5.1.1 Java ME Architecture

The architectures of the different Java implementations are shown in Figure 5.1. As shown in the figure, Java is available for several different environments and usages, stretching from Java EE as the most resource-demanding platform to the JavaCard API, being a very lightweight platform.

The Java ME architecture is divided into two main parts, shown as columns in the figure, based on the complexity of the systems on which they are designed to run. Whereas the left column is meant to be used with high-end consumer devices, such as smartphones and PDAs, the right column is meant for the typical, resource-weak mobile phones.

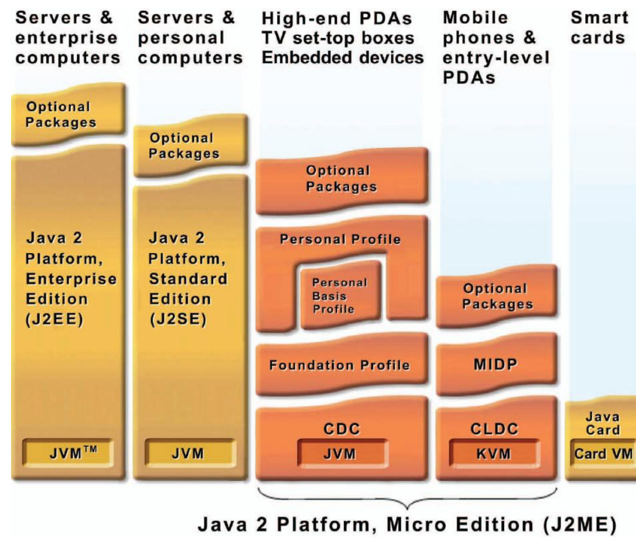

Figure 5.1: Java ME architecture

In our project, we will focus on developing games for the mobile phones used by most people today. The majority of these phones fall into the category "Mobile phones and entry-level PDAs". Because of this, the rightmost of the two columns labeled Java ME in the figure is the one most interesting for our project. The following sections will explain the layers for low-end mobile devices shown in the figure, as well as explain a few central concepts.

Java Virtual Machine

A Java Virtual Machine (JVM) is an execution engine for Java applications, which has predefined machine instructions. When compiling Java source code, the code is translated into byte code which then is executed in the JVM. This separates Java from other programming languages such as C++ in that these languages are compiled directly into machine code.

For the Java ME version designed for resource-poor devices, a specific virtual machine called the Kilobyte Virtual Machine (KVM) has been designed. The goal when designing the KVM was to create the smallest possible virtual machine that would still maintain all the central aspects of the Java language [13]. As a result of this, the KVM has been stripped of much of the functionality offered in the standard JVM [24].

Configurations

A configuration in Java ME is supposed to represent the minimum platform for its target device [56]. The configuration is not allowed to offer optional features. Because of this, all Java ME implementations using the same configuration offer exactly the same basic functionality, which makes a solid foundation for cross-platform development, and also keeps the size of the configuration at a minimum.

The configuration specified for the low-end Java ME implementation is the Connected Limited Device Configuration (CLDC). This configuration is designed to run on top of the KVM. The most current version of CLDC is version 1.1 (JSR-139). Compared to version 1.0, CLDC

1.1 includes features such as floating point and weak reference support, in addition to other enhancements.

Profiles

The profiles complement the configuration by adding more specific APIs to make a complete runtime environment for running applications. For the CLDC, the Mobile Information Device Profile (MIDP) is the profile best suited for standard mobile phones. This profile adds networking, user interface components and local storage to CLDC [56]. In addition, the Information Module Profile (IMP) is implemented for embedded, "headless" devices, such as vending machines and other devices with limited or no display, and limited network connectivity.

The newest version of MIDP, version 2.0, introduced the `javax.microedition.lcdui.game` package, which is specially designed for game development for Java ME. Some of the new features supported by this package are collision detection, sprites, tiled backgrounds, layers, and layer management. These features simplify the process of developing 2D games for mobile phones [63]. MIDP 2.0 is fully backwards compatible with MIDP 1.0.

Additional Specifications

In addition to the KVM, CLDC, and MIDP layers, Java ME supports a number of additional specifications, which offer functionality beyond that offered in the standard Java ME packages. These specifications are described in Java Specification Requests (JSR), which can be used for implementation of the technology specified. Work is continuously being done to improve existing, and implement new, such specifications. Some of the most important of these specifications are:

- JSR-82: Java APIs for Bluetooth
- JSR-118: Mobile Information Device Profile 2.0
- JSR-139: Connected Limited Device Configuration 1.1
- JSR-172: Java ME Web Services Specification
- JSR-177: Security and Trust Services APIs
- JSR-234: Java advanced multimedia supplements for Java ME
- JSR-239: Java Binding for the OpenGL ES
- JSR-293: Location API 2.0 for Java ME

MIDlets

A Java application written for Java ME and MIDP is called a MIDlet. A MIDlet consists of at least one class implementing the `javax.microedition.midlet.MIDlet` abstract class. This class then becomes the startup class of the application (analogous to the class implementing the `main()` method in Java SE applications).

In order to create a MIDlet, the program files need to be packaged in a Java Archive (JAR) file. In addition, to enable distribution of third party MIDlets, the developers need to create a Java Application Descriptor (JAD) file. This file contains meta data used by the Java Application Manager (JAM) to verify and configure the MIDlet at runtime.

5.1.2 Limitations

Java ME is not a platform for developing advanced and complex applications neither when it comes to the application size nor resource demands. It lacks many of the features found in programming languages like C++ or standard Java. As an effect of this, it is important to be realistic about what can and what can not be included in a Java ME MIDlet.

The number of optional packages left for the manufacturers of mobile phones to include or exclude also leads to different bases for running Java ME applications on different mobile phones. Applications that only make use of the standard functionality found in CLDC 1.0 and MIDP 1.0 should be supported by most mobile phones. However, once a developer needs to include functionality found in other specifications, he needs to control whether this specification is supported by the mobile phone he is developing for or not.

5.2 Mobile Phones

The goal of this project is, as previously mentioned, to develop a client-server multiplayer mobile game. When developing a mobile application, one always has to be aware that different phones offer different functionality. To be certain that an application or game works as it is supposed to, it is important to be able to test it on as many different mobile phones as possible.

5.2.1 Features

Today's mobile phones offer a variety of different features and functions, where some are accepted as standard functionality for a mobile phone, whereas others are specific for certain phone models. This section discusses some of these features and functions, and what effects these may have on mobile gaming. This section is based on our depth study [29] and has been rewritten and modified to suit the different project scope.

Keypad

The dials on most mobile phone models is one of the greatest challenges when it comes to developing mobile applications. While computers have more than a hundred keys, most mobile models only offer between ten and fifteen keys, and usually a four or five directional navigation key. This forces developers of mobile applications to think carefully what each key and combination of keys should do, as illogical or complex keypad layouts may lead to the users avoiding the application.

Within game development, developers of console games have had to work with relatively few keys ever since the first gaming consoles. Even though today's controllers have more keys than the first ones had, the consoles still do not have more keys than mobile phones. It may therefore be wise to look at console games for inspiration on how to find good key layouts for mobile games.

Vibration

Vibration is a feature supported by most new mobile phone models today. This can be used for notifying the user when something important occurs, such as a collision. A good example of

how vibration can increase the gaming experience is the introduction of vibrating controllers on the Sony Playstation 2 and Microsoft Xbox consoles. Vibrating controllers extend the gameplay as players get additional feedback to their actions than just visual feedback. Vibration could be implemented when two or more players collide or when other game events occur. The additional feedback will also help players comprehend that game events occurred.

Screen

Some kind of screen is available on all mobile phone models. Today's mobile phone screens usually come with a number of displayable colors and a screen resolution more than sufficient for reading text messages and navigating menus. However, when it comes to advanced mobile applications and games, the resolution of the screen in particular represents a considerable challenge. For the user to be able to see the contents on the screen, the details need to occupy a sufficient amount of pixels, which limits the number of possible visible objects at once. Because of this, mobile game developers need to find ways to display enough, but not too much, information at any time.

Touchscreen

A few mobile phone models offer a pressure sensitive screen, which reduces the problem of few available keys as actions then can be performed when the user touches specific areas of the screen. In addition, a touchscreen has the possibility of tracking the user's finger across the screen. This opens up for more complex and smooth movement in a game than what is possible when only using the keys. However, this also means more complex implementation of player movement in the development.

Unfortunately, touchscreen is only available on very few mobile phone models. In addition, those models are primarily business phones, and not very often bought by typical mobile gamers. Because of these factors, developing a mobile game requiring a touchscreen will most likely not be successful because of the very small target group.

Internet Connection

Internet connection is included in nearly all mobile phones today, as described in Section 5.3. This can be used to send and receive information from other mobile phones over the Internet or from a server. Depending on the mobile network supported by the mobile phone, the speed of file transfer can be very slow (if the mobile phone only supports GSM) or fast (if the mobile phone supports UMTS) (Section 5.3). For our project, the Internet connection is essential, since the type of mobile games we are considering use this connection for their communication.

Internet connection may also be used to add extra content to the game, by integrating information found on the Internet into the game. This could make the game very flexible, and lead to every game session being a little different from the game session before. Also, extra content like graphics, maps, rules, and other updates may be downloaded from the Internet. This would extend the game's longevity.

5.2.2 Requirements for Mobile Phones in this Project

To be able to test our prototype games, a few requirements will have to be fulfilled. These requirements are:

- Java ME support, containing these packages:
  - JSR-118: MIDP 2.0.
  - JSR-139: CLDC 1.1.
- Support for one or more of these network technologies:
  - General Packet Radio Services (GPRS)
  - Enhanced Data rates for GSM Evolution (EDGE)
  - Universal Mobile Telecommunications System (UMTS)
  - Wireless LAN (WLAN)
- At least 80x120 pixels screen resolution.

5.2.3 Our Test Phones

Table 5.1 lists the mobile phones we have available for testing our MIDlets, and the phones' specifications. In the table, SE is an abbreviation for Sony Ericsson. The fields used in the table are:

Model: The manufacturer and model name of the mobile phone.

Resolution: The resolution of the phone's screen.

MIDP: The latest version of MIDP supported by the phone.

CLDC: The latest version of CLDC supported by the phone.

GPRS: Whether the phone supports GPRS or not

EDGE: Whether the phone supports EDGE or not

UMTS: Whether the phone supports UMTS or not

WLAN: Whether the phone supports WLAN or not

Table 5.1: Available test phones

Model	Resolution	MIDP	CLDC	GPRS	EDGE	UMTS	WLAN
SE K610i	176x220	2.0	1.1	✔	✘	✔	✘
SE K750i	176x220	2.0	1.1	✔	✘	✘	✘
SE K800i	240x320	2.0	1.1	✔	✘	✔	✘
SE W850i	240x320	2.0	1.1	✔	✘	✔	✘
Nokia N70	176x208	2.0	1.1	✔	✔	✔	✘
Nokia N80	352x416	2.0	1.1	✔	✔	✔	✔

When developing applications for mobile phones, it is important to bear in mind that they offer far less power in terms of memory and processing than computers [12]. Extensive testing on all mobile phones that are to be supported is therefore very important to ensure the compatibility.

For this reason, Table 5.1 should be expanded with all new mobile phones used for testing. Phones not found in this table can not with certainty be said to be able to run the application.

5.2.4 Emulators

An emulator is a software application that simulates the hardware of a given system so that different hardware configurations may be tested on one single computer. Within Java ME development, an emulator is an application that simulates a mobile phone so that Java ME MIDlets can be tested without actually having to transfer them to a mobile phone. This is very useful as it reduces implementation time, and also is a good indicator on whether or not the MIDlet will work on a given mobile phone model.

Sun provides a default mobile phone emulator in their standard JWT, which may be configured to simulate mobile phones with different specifications. In addition, most mobile phone manufacturers offer emulators for their models. These emulators usually are more accurate in their imitation of the specific models. However, it is important to bear in mind that the emulators are not the actual phones, and may have minor (or major) deviations from the real phones' behavior.

5.3 Mobile Network Technologies

Mobile network technologies are the technologies used for communication and data transfer with mobile phones. Mobile networks are usually categorized in generations depending on when they were deployed and their technological complexity. The technological progress has followed the increasing number of mobile phone users and the expected number of services provided by a mobile phone. Also, alongside the mobile network technical progress the number of Internet users has expanded immensely. Internet access with a mobile phone and related multimedia services are pushing the development of mobile network technologies. The following sections describe current mobile networks, compare them to each other, and present some future mobile networks.

5.3.1 2G Networks

Second-Generation Technology (2G) consists of mobile communications technologies that uses digital radio signals instead of analog in a circuit-switched domain. 2G also uses digital communications between the network entities [31]. 2G technologies can be divided into TDMA- and CDMA-based standards depending on the multiplexing types used. With First-Generation Technology (1G) there was little standardization. This lead to national standards and little support for roaming, or switching from one network to another. With 2G's semi-global standards, the goal was to increase roaming, as well as increase the quality of voice services compared to 1G.

The advantages of 2G are increased sound quality due to error checking, decreased power consumption due to digital signals, digital services like SMS and e-mail, and increased system capacity. The increase in system capacity is because of more effective compression and multiplexing with the use of digital data. The limitations and disadvantages of 2G networks are low transfer rate, limited roaming due to multiple standards, and low efficiency for packet-switched services [31]. These limitations makes pure 2G networks unsuitable for the services

users demand today like wireless Internet access and high data rates necessary for video calls, downloading of data, or gaming.

GSM

The Global System for Mobile Communications (GSM) is an open, digital cellular technology used for transmitting mobile voice and data services. It is considered to be the most popular standard for mobile phones in the world with over 2 billion users worldwide [21]. GSM is a circuit-switched system with 200 kHz channels divided in 8 time-slots of 25 kHz each and uses time division multiple access transmission methods. Basic data services like SMS are supported with data transfer speeds of up to 9,6 kbit/s. This also increases the digital voice quality compared to previous standards. Along with this and other technological advances GSM's standardization of subsystem interfaces, it has provided flexibility in manufacturers' and network operators' development work and configurations [59].

GPRS

General Packet Radio Services (GPRS) is a mobile data service often described as a 2.5G service, which means that it is placed between 2G and 3G services. This is because it does not provide faster services than a 2G service, but it uses packet-switching like a 3G service. Its main objective is to offer access to standard data networks such as TCP/IP and X.25 [22]. GPRS is packet-switched, which means that multiple users share the same transmission channel, only transmitting when they have data to send. This procedure is called an request-allocation procedure. The system is billed per megabyte of data transfer, instead of per minute of connection time like a circuit-switched system is. Thus, packet-switched data transmission is cheaper than circuit-switched data transmission. Because of the mentioned request-allocation procedure, users experience the system as "always on" [36], which is an advantage with GPRS. However, it uses the same modulation as "regular" GSM data transmission.

GPRS can be utilized for data services like MMS, instant messaging, Internet applications that uses WAP, and Push-To-Talk (PTT). GPRS' use of packed-switched connections delivers a much shorter access time to the network compared to circuit-switched connections [22]. The GPRS mobile station (base station) can use between 1 and 8 time slots over the same basic air interface as used in GSM [36]. This air interface consist of a 200-kHz channel that is divided into 8 time slots. These time slots are dynamically allocated when there are packets to send or receive. Uplink and downlink channels are reserved separately, thus various combinations of channels are possible. The communication with GPRS can be divided into two categories; Point-To-Point (PTP) or Point-To-Multipoint (PTM) [59, 22]. The PTP service transmits single packets between two users. The PTM service supports transmission of data packets between a user and a specified group in a certain geographical area.

GPRS has 4 different coding schemes that delivers different levels of robustness and transmission rate per time slot. CS-1 is the most robust scheme, while CS-4 is the fastest and least robust scheme, i.e. if the coverage is good, this scheme can deliver higher data rate. Newer devices can switch between the schemes depending on the coverage at their location. The maximum throughput transmission rate ranges from 32-40 kilobits per second (kbit/s) depending on the mobile phone and the coding scheme, but the theoretical maximum throughput is 160 kbit/s per mobile station when using 8 time slots without error correction [22]. The latency of GPRS is very high, as the round trip message time (from server to client and back) is in the order of 700-1000 milliseconds, with a standard deviation time of 150 milliseconds [40].

EDGE

Enhanced Data rates for GSM Evolution (EDGE) is a digital mobile phone technology designed to increase the data transmission rate and improve transmission reliability. It is unofficially considered to be a 2.75G service since it has a packet-switched domain, but does not provide the required data rate to be classified as a 3G service. The technology is suitable for both circuit- and packet-switched services. Enhanced Circuit-Switch Data (ECSD) covers the circuit-oriented part and Enhanced General Packet Radio Services (EGPRS) covers the packet-oriented part [59]. EGPRS has an increased data capacity compared to GPRS. According to the Global mobile Suppliers Association (GSA) [20], as of February 1. 2007 there were 196 commercial deployments of GSM/EDGE in 105 countries.

EDGE requires mobile station modification since existing GSM mobile stations do not support the new modulation technologies [36]. However, no software or hardware modification on the devices is needed as long as the device has GPRS functionality implemented [22]. It can be used for any packet-switched applications like an Internet connected application or for the services mentioned in the GPRS section, because EDGE delivers an enhancement of GPRS with EGPRS. EDGE can deliver throughput transmission rate up to 236,8 kbit/s for 4 time slots in packet mode, and the theoretical maximum throughput is 554 kbit/s for 8 time slots [59].

The reason for EDGE's enhanced throughput is that the amount of data sent per signal is tripled compared to GPRS. This means that EDGE is not faster than GPRS, but transfer more data each time, which results in a higher data rate [22]. This enhancement is due to the introduction of Octagonal Phase Shift Key (8-PSK) modulation. The 8-PSK signal is able to carry 3 bits per modulated symbol as opposed to the Gaussian Minimum Shift Key (GMSK) signal's ability to carry just 1 bit.

5.3.2 3G Networks

Third-Generation Technology (3G) networks are developed under the International Telecommunication Union (ITU) initiative. ITU have defined 3G as *"a term coined by the global cellular community to indicate the next generation of mobile service capabilities in terms of bandwidth and network functions. These service capabilities in turn allow advanced services and applications, including multimedia"* [59].

When the 3G hype was introduced, the possibility for using video calls was expected to be the "killer app" convincing mobile users to purchase 3G enabled mobile phones. However, video calls have not reached its expected popularity, but other characteristics offered through the 3G technologies still have proved 3G to be a useful and sometimes necessary development. Among these characteristics are new services with high quality of service, high capacity, high spectral efficiency, and high security [59].

The world's first commercial 3G service, FOMA, was launched by NTT DoCoMo in Japan in October 2001 [59]. In later years, 3G networks have also spread to other countries, and in December 2004, Telenor launched Norway's first UMTS network [52]. The evolution of the 3G system inside the Third Generation Partnership Project (3GPP) has been organized and scheduled in phases and releases (99, 4, 5, and 6) [3].

The collective term 3G cover several different technologies. In Norway, the 3G technology currently in use is UMTS. In short time, this technology will be supplemented with another

3G technology: High-Speed Packet Access (HSPA). These two technologies are described in the subsequent sections.

UMTS

Until recently, Universal Mobile Telecommunications System (UMTS) has been the only available 3G technology in Norway, and it is supported both by Telenor's and by NetCom's mobile networks. Using the UMTS technology with a R99 handset (compatible with 3GPP release 99), one can expect a data transfer speed of up to 384 kbit/s. However, as the available bandwidth depends on the density of nearby antennas and their connected users, the users may often experience lower actual transfer speed than this [49]. Still, compared to GSM networks, UMTS offer a great improvement considering the available data transfer speed.

In 1998, two modes were specified for the UMTS air interface, UMTS Terrestrial Radio Access (UTRA). The first of these is based on paired bands with Frequency Division Duplex (FDD) transmission, whereas the other operates in a single band using Time Division Duplex (TDD) transmission. For their transmission, these modes use Wideband CDMA (W-CDMA) and Time Division CDMA (TD-CDMA), respectively [59, 3]. Of the two modes, UMTS over W-CDMA is currently most used. Here, two 5 MHz channels are used for uplink and downlink. Like GSM using GPRS or EDGE, UMTS contain a circuit switched and a packet switched domain. The circuit switched domain provides the services related to voice transfer, whereas the packet switched domain provides those related to data transfer [31].

Between each packet of data transmitted, a Transmit Time Interval (TTI) is specified. For UMTS, this interval can take the values of 10, 20, 40 or 80 ms. For voice services, the TTI is fixed at 10 ms, whereas it changes according to the services used for data services [3]. In practice, this means that no more than a maximum of 100 packets can be sent each second using UMTS.

The UMTS air interface is in itself incompatible with the GSM network, and in most cases, these two networks operate in different frequency bands. The standard UMTS frequency bands as defined by the 3GPP are 1885-2025 MHz for uplink, and 2110-2200 MHz for downlink. However, in some countries, other frequency bands are used instead of these [59]. Most mobile telephones that are sold with 3G support today are hybrid phones that support both GSM and UMTS, so that they can utilize the most suitable available network anywhere.

UMTS support a number of services in addition to those already found in GSM and its supplementary networks. In particular, the Multimedia and Interactive Multimedia services are services that require high bandwidth and are attractive for the average, everyday mobile users. In addition, the UMTS specification support teleservices and applications like paging, database inquiries, electronic mail, and teleshopping [59].

HSPA

High-Speed Packet Access (HSPA) was the main improvement in release 5 of the 3G standards, and significantly increases the download and upload speeds of UMTS. HSPA can be implemented in UMTS's standard 5 MHz carrier, and co-exist with the first generation of UMTS networks. It is therefore an extension of UMTS rather than a brand new technology.

The HSPA acronym is a generic term referring to improvements made both to the downlink (HSDPA) and uplink (HSUPA) channels. High-Speed Downlink Packet Access (HSDPA) offer far greater data transfer speed than UMTS, with a theoretical speed of 14.4 Mbit/s, and an

expected speed of 3.6 Mbit/s. With transfer speeds like this, HSPA equals or exceeds fixed networks like ADSL, and offers mobile users great possibilities for services like streaming media [3].

The enhancements of HSDPA result from a number of new technical capabilities to the radio network, which when combined offer a significant improvement for both end users and operators. Among these capabilities are a common shared downlink channel (HS-DSCH), which can be simultaneously used by multiple users, and a shorter TTI (2 ms), which enables higher speed transmission [1].

Across Europe, HSPA is currently being introduced as UMTS' successor. It is popularly called 3.5G, or "super-3G", because of its significant improvements compared to the original UMTS technology. In Norway, HSPA has yet to be widely offered to the mobile users, but Norway's two largest network operators, Telenor and NetCom, have both indicated their interest for the technology. As of June 2007, NetCom have released HSDPA support in their mobile networks in limited areas, whereas Telenor are testing HSDPA in their laboratories [57]. High-Speed Uplink Packet Access (HSUPA) support will not yet be provided, but HSDPA by itself also greatly increase the speed of the networks.

5.3.3 WLAN/WiFi

A mobile phone user is dependent of always having an available network so that he can be reached, and is able to reach others. Both 2G and 3G networks have been developed to meet this requirement, so that the users are always within reach of a transmission antenna. However, other networks technologies exist that may be used within limited areas, and may increase the transfer speed to and from the mobile phone significantly. One such kind of network technology is the WLAN, or WiFi technology.

The 802.11 family is a set of WLAN standards defined by IEEE. The goal of these standards is to provide wireless LAN services that are consistent with 802.3 Ethernet networks. Today, three of these standards are in wide use: *a*, *b* and *g*. These three standards all use the same protocols for communication, but have somewhat different properties in terms of use of frequency channel and transfer speed. Table 5.2 shows the properties of the 802.11x standards [36].

Table 5.2: Current 802.11x WLAN standards

Name	Channel	Speed (theory / expected)	Range (in / out)
802.11a	5 GHz	54 Mbit/s / 25 Mbit/s	~25 m / ~75 m
802.11b	2.4 GHz	11 Mbit/s / 6.5 Mbit/s	~35 m / ~100 m
802.11g	2.4 GHz	54 Mbit/s / 25 Mbit/s	~25 m / ~75 m

As seen in the table, both 802.11b and 802.11g make use of the 2.4 GHz frequency band for their transmission. Because of this, 802.11g is backwards compatible, that is, an 802.11g compatible unit can be used in an 802.11b network.

802.11 networks have normally been used for computer networks, supporting both infrastructured and ad-hoc networks (with and without a central access point). However, some mobile phones also support 802.11 networks, and where such are available, the data transfer may be significantly improved by connecting to a WLAN network. For a 3G mobile phone supporting UMTS, the download speed may be increased more than 50 times by connecting to a 802.11g network instead of using UMTS. However, it is important to notice that for Internet communication, the actual transfer speed can never exceed that of the WLAN's own connection to the

Internet. In practice, this means that a WLAN connected to the Internet through a ADSL line with a download speed of 2 Mbit/s can not offer a mobile user more than maximum 2 Mbit/s download speed outside the local network, even though the WLAN itself may be an 802.11g network.

5.3.4 Comparison

Table 5.3 compares the network technologies discussed in the above sections. The technologies are compared by expected practical transfer speed/data rate, practical latency, the cost of downloading data with this technology with a Telenor [53] mobile subscription, power consumption, and the deployment of the technology both on mobile phones and with carriers.

Table 5.3: Comparison of mobile network technologies

Network	Transfer speed	Latency	Cost	Power	Deployment
GSM	14,4 kbit/s	High	-	Low	High/High
GPRS	40 kbit/s	High	20 NOK/MB	Low	High/High
EDGE	160 kbit/s	Medium	20 NOK/MB	Medium	Low/High
UMTS	384 kbit/s	Medium	20 NOK/MB	Medium	Medium/High
HSPA	3,6 Mbit/s	Medium	-	Medium	Medium/Low
WLAN	25 Mbit/s	Low	0 NOK/MB[1]	High	Low/[2]

[1] Some commercial WLANs charge per hour used.

[2] WLANs are provided by private individuals, or by commercial WLAN providers that range from mobile phone carriers to hotels or airports.

Latency, transfer speed, and deployment are the most important aspects for real time multiplayer mobile games. The amount of latency and transfer speed determines how often clients will send and receive data updates to and from the server. Low latency and high transfer speed are important to prevent lag or delays in the game. Less lag will give the game a smoother gameplay with better player movement. The degree of deployment determines the size of the user group. The larger the mobile phone deployment is, the better the technology is suited for this project. For instance, a game based on a technology that is supported by only a few mobile phones will not reach many users. Cost and power consumption have less impact on the development of a multiplayer mobile game. However, these aspects are important to users since they do not want to pay for a game with an exaggerated cost and limited playability.

5.3.5 Other Networks

The previous sections consider mobile networks that exist and are available for mobile phone users in Norway today. For this project, these are the most interesting networks for testing and evaluating mobile games. However, a number of networks are under development and will be available in the future. Multiplayer mobile game developers should therefore be aware of these networks. This section offers short introductions to the most important of these future networks.

4G

Fourth-Generation Technology (4G) have no set definition since the 4G technologies has not yet been fully developed, tested, or implemented. The only requirement to such technologies is that the technologies should be beyond any 3G technology both in bandwidth and data throughput.

4G does not yet have a killer application, though the improved bandwidths and data throughput offered by 4G networks should provide opportunities for previously impossible products and services. Examples of such products or services could be streaming high-definition television, or downloading full featured movies.

802.11n

The 802.11n standard is the latest member of the 802.11 family. As of today, only propositions for the standard have been released, and the official standard has not yet been decided upon. However, a number of products based on the proposed standard are available for purchase. These products offer the specifications shown in Table 5.4. For ease of comparison with the other 802.11 standards listed in Table 5.2, this table is presented correspondingly.

Table 5.4: Proposed 802.11n specifications

Name	Channel	Speed (theory / expected)	Range (in / out)
802.11n	2.4 GHz / 5 GHz	540 Mbit/s / 200 Mbit/s	~50 m / ~125 m

WiMax

Another standard with IEEE is the 802.16 standard, officially called Wireless MAN (WMAN), or more commonly known as WiMax. Like the 802.11 family, WiMax is a standard for high-speed data transfer between network entities. However, WiMax support higher performance both in terms of transfer speed and network range than what is the case with the WLAN standards. This makes WiMax networks very suitable for urban areas.

The theoretical range for a WiMax network is as much as 112.6 km, but this requires perfect conditions. Also, in perfect conditions, the theoretical transfer speed for WiMax is 70 Mbit/s. However, the available transfer speed is reduced as the distance between the transmitter and the receiver is increased. In practice, if there is a direct line of sight between the transmitter and receiver, one can achieve a transfer speed of 10 Mbit/s over 10 km. For urban areas, where the line of sight is often obstructed by tall buildings, the practical range and speed is 10 Mbit/s over 2 km.

5.4 Transport Protocols

Transport protocols are used in the Transport Layer of the Internet reference model. These protocols deliver data from one application running on the Internet to another [38]. They specify source and destination port numbers used to locate the correct end point for both the sender and the receiver. Transport protocols can either be connection-oriented (the protocol establish an end-to-end connection before data is sent) or connectionless (data is sent to an

address without the protocol checking if the recipient is ready or connected). Connectionless protocols are also defined as stateless since the endpoints do not have the possibility to remember where they are in the message exchange. Connection-oriented protocols on the other hand, do remember this, and are thus also called stateful. Connection-oriented protocols guarantee that data will arrive in the correct order. They are therefore considered reliable network services. Connection-less protocols will have more frequent problems with sending data, thus it is necessary to resend data more often than with connection-oriented protocols if all packets must be received.

5.4.1 TCP

Transport Control Protocol (TCP) is the dominant transport layer protocol in use today, providing a reliable in-order stream of data between two applications [38]. A TCP connection is a connection between only two endpoints. The protocol turns a sequence of application writes into a reliable, in-order stream of bytes. If packets are lost, delayed, or changed TCP will detect this and retransmit the packets. To ensure this reliability, information about both the packet and the sequence is sent in the packet header. The header also contains an acknowledge number used to control that the correct packet is received in the correct sequence [17]. To detect corruption of data in transit, a checksum field in the header is used. The checksum is a number derived by the sender using a mathematical function on the header. Then, the receiver derives a comparison number by using the data received and the same function. If the checksum and the comparison number does not match, the packet is discarded and has to be resent.

Other features with TCP is that it is full-duplex, i.e. that each endpoint can be both sender and receiver simultaneously. It also provides flow-control and congestion avoidance. Flow control is the ability for a receiver to slow down the sending rate to avoid overwhelming the receiver with data and thereby wasting resources. Congestion avoidance is used to limit the sending rate in response to network congestion.

5.4.2 UDP

User Datagram Protocol (UDP) is a much simpler protocol than TCP, containing fewer features [38]. It sends datagrams in chunks and guarantees whole packets at arrival. However, it only provides a small amount of extra functionality over the network layer protocol (Internet Protocol (IP)). An indication of this is the small UDP header [16]. It only contains source and destination port numbers, a checksum for error detection, and the length of the datagram. Since UDP is connectionless and packet-switched, it does not provide a reliable, in-order delivery. With no support for end-to-end connection, UDP does not need to perform a three-way handshake to set up the connection, which results in less overhead. Instead data can be sent immediately. The protocol also supports broadcast or multicast for transmission to multiple recipients at the same time.

Much of the TCP functionality relies on receiver feedback, as the sender uses the acknowledge numbers to determine what messages to retransmit. UDP however has no support for receiver feedback besides sending a datagram from the receiver to the sender. This means that the application must control discovery of lost packets and retransmission itself. UDP also does not check if the packets have arrived and just assumes that they are received when sent. With less overhead, more lightweight setup, and less functionality, UDP is considered faster and

more efficient, but less reliable, than TCP. To increase the reliability and other important properties, TCP features can be implemented on top of UDP by the application.

5.4.3 SCTP

Stream Control Transmission Protocol (SCTP) is a unicast protocol, and supports data exchange between exactly two endpoints. The protocol provides reliable transmission, detecting when data is discarded, reordered, duplicated or corrupted, and retransmitting damaged data as necessary. The transmission is full duplex and message oriented. Messages can be bundled into messages with individual message boundaries. In comparison, TCP is byte oriented and does not preserve any implicit structure within a transmitted byte stream without enhancement [19]. Messages are assigned with a Transmission Sequence Number (TSN) and the receiving end acknowledges all TSNs received even with gaps in the sequence [18]. This ensures that messages are received in-sequence and that the lost messages are retransmitted.

SCTP and TCP have several resembling features like flood control and congestion avoidance. However, SCTP is more complex than TCP due to improved error detection and security. Also, it has the capability to transmit several independent streams of messages at the same time, i.e. it is a multi-streaming protocol. SCTP is designed to be used in situations where reliability and near-real-time considerations are important.

5.4.4 Comparison

Table 5.5 compares the transport protocols mentioned above. The comparison is focused on important aspects for transport protocols and the terms are explained in Section 5.4. The protocols' packet header size shows how many bytes in each packet are reserved for "non-data". The header is used for giving destination port, checksum for error detection, length of packet, and other important information needed to receive the packet correctly.

Table 5.5: Transport protocols

	TCP	**UDP**	**SCTP**
Packet header size	20 bytes	8 bytes	12 bytes
Packet entity	Segment	Datagram	Message
Error checking	Yes	Yes	Yes
Port numbering	Yes	Yes	Yes
Connection oriented	Yes	No	Yes
Automatic repeat request	Yes	No	Yes
Segment numbering	Yes	No	Yes
Flow control	Yes	No	Yes
Congestion avoidance	Yes	No	Yes

Because of SCTP's complexity and small deployment, we choose to develop and test with TCP and UDP as the transport protocols. This will both test the benefits and trade offs with segments and datagrams, connection orientation, and packet header size. However, all of TCP's features and extensive functionality can be implemented by an application over UDP. Thus, if some of these features are needed or desired, they can be implemented. This will reduce the overhead compared to TCP, yet deliver the needed functionality.

Chapter 6

State-of-the-Art

This chapter describes state-of-the-art aspects relevant for this project. Some of the mobile games mentioned in this chapter has been discussed in our depth study report [29] and therefore some of the text is based on that report. However, because of a different focus in this project, new elements and aspects have been explored.

6.1 Today's Multiplayer Mobile Games

Because of the rapid evolution of mobile network technology and their availability in conjunction with the mobile phones' increased processing powers, improved power management, and better screens, satisfying multiplayer games for mobile phones are now possible. Multiplayer games, or games with multiplayer game modes, have always had an extra appeal because of the interaction with other players. This interaction has to be stable, reliable, and need to be perceived as fluent by the player. The perception of fast player interaction will make the gameplay feel more fluent and dynamic. Also, graphics that are interesting, good-looking, and easy to comprehend are needed to attract players. In the next chapter a selection of todays multiplayer mobile games are presented, including brief sections about the games' business models and their used network technology.

6.1.1 Pirates of the Caribbean Multiplayer Mobile

Based on the Disney movies *"Pirates of the Caribbean"*, this game is developed by Floodgate Entertainment [11] and published by mDisney Studios [39], the mobile entertainment department of Disney. It claims to be the second available MMMG and real-time multiplayer mobile phone game. The game runs in Binary Runtime Environment for Wireless (BREW), the counterpart to Java ME's KVM [44, 2]. Up to 16 players can compete and/or collaborate with their ships in gameplay instances, as shown in Figure 6.1. The interaction between players (maneuvering around and shooting other players) is real-time and the game uses masking delays techniques to conceal network lag. These techniques mean that the games uses graphical methods to make the game look more fluent. For instance, the ships must turn around gradually and do not change direction on the spot. This means that position updates can be handled smoother and movement prediction can be utilized better.

Each player has control over one of three different types of ships, which can be used in three game modes: a basic combat mode with respawning, a capture the flag mode, and an assault/defend mode. All three modes are played by two teams with 2 to 8 players on each team. Players can form guilds (group of players that regularly play together) and challenge other guilds for control and domination of certain areas in the game world. Players can also choose between three different ships with different stats in terms of speed, armor, and firepower.

Figure 6.1: Screenshot of Pirates of Caribbean Multiplayer Mobile Game (2006) from Floodgate Entertainment

The graphics of the game include visual effects like waves hitting the shore, ships recoiling after shooting their cannons, and smoke effects. The game also delivers a chat room area where players can discuss the game or just socialize. Communication between players is supported both in-game and outside of it. This provides great communication possibilities and help the game community, which is further supported with a website containing forums and leaderboards. By integrating the game's website into the game, the game community is further extended. The website dynamically reflects the game and updates the leader boards and other lists related to the game. By completing missions, sinking ships, and winning battles, players gain "infamy", which is what ranks the players and guilds on the leaderboards. Also, infamy is used to level up the players by increasing the ship speed or decreasing the cannon reloading time.

Business Model

The game was released in July 2006 and is only available on one major US wireless carrier service (Verizon, [65]) as a subscription-based game. The monthly subscription is $3.99, but the users must pay for the airtime, i.e. the download and upload time, they use to play the game [64]. To keep the gameplay and the play environment fresh, challenging and evolving, new maps, new alliances, new power ups, and new missions are added to the game frequently. These additions are available for download by using WAP.

Network Technology

The game is only available on Verizon Wireless supported mobile phones since the game uses Verizon Wireless' *V CAST* service [64, 66]. *V CAST* is a 3G Evolution-Data Optimized (EV-

DO) network used for streaming video and music clips as well as playing games. *V CAST* is implemented as a BREW application and has a download speed of between 400 and 700 kbits/s. EV-DO is a wireless radio broadband data standard not available in Norway, but used by many major service providers in North America and Asia. It is classified in the Code Division Multiple Access (CDMA) family of standards and is significantly faster than EDGE.

Pirates of the Caribbean Multiplayer Mobile has a client-server architecture. The server creates instances players can play in and saves players' scores as well as update the leaderboards. Since the game is an action game and requires frequent data updates, the latency needs to be low and the data updates must happen close to real time. With *V CAST*'s fast Internet connection these aspects are well taken care of.

6.1.2 Samurai Romanesque

Samurai Romanesque is a role-playing game developed in Java ME. Players interact in a world including martial arts, adventurous travel, Zen riddles, and romance in 15th-century Japan [28]. The game consist of three applications: a training application to learn martial art skills that introduce the player to the game, a multiplayer application to participate in the game, and a chat application to receive assignments from the server and to communicate with other players. The training application also has three separate parts: sword training, physical strength, and mind training. Each of these parts consists of minigames, where the sword minigames increase the players hand-eye coordination and the mind minigames test the player's patience and memory. In the physical strength minigames the player takes a part-time job with various task such as moving items. This job pays money and increases the player character's strength. The player's performance in all the training minigames are tracked with a point system, which is used for improving the skills of the player's character.

The players can play the game with three different motivations: fame, career, and love. The player gets famous by winning battles, he can rise in the samurai ranks that can lead to the player becoming a warlord, and he can try to get a wife by rescuing a damsel in distress. The life of a virtual samurai is limited to 40 days, after that period the samurai dies. However, by obtaining a wife, the player can "produce" a son that inherits his father's status and score, which lets the player continue the game. Another element in the game enhancing the realism level is that the weather in the game's regions are real-time representation of the real weather conditions in the same region.

NTT DoCoMo [10] uses a version of Java ME called Internet Applications (i-Appli), which supports graphics-based functions like GIF, horizontal scrolling, and list boxes. The game takes advantage of this, using three layers to compose images. The layers shows the landscape, which is scrollable as shown in Figure 6.2, the character's clothing, which is determined by the role the player chooses, the character's face, which can contain scars from battle, and the hairstyle of the character, which can be changed by visiting barbershops. The character's face is used in the chat application to represent the players. The game consists of more than 300 location maps with towns, including places where the players can obtain information, earn money, or purchase items. These maps are the horizontally scrollable landscapes mentioned as layers.

Business Model

Samurai Romanesque was released in Japan in 2001 on the NTT DoCoMo packet-switched i-Mode network as an i-Appli. Content providers usually charges between $1.00 to $2.50 in

Figure 6.2: Screenshot of Samurai Romanesque (2001) from Dwango

subscription fee with NTT DoCoMo retaining 9 percent of the net income. Apart from subscription fees, users pay 2 cents per received data packet, each with a size of 128 bytes.

Network Technology

NTT DoCoMo's 2.5G-network speed has increased from 9.6 to 28.8 kilobits per second. NTT DoCoMo also provides a 3G service, Freedom of Mobile Multimedia Access (FOMA), that offers a download speed of 384 kilobits per second. This service enables streaming video (movie trailers), and live videoconferencing. However, Samurai Romanesque uses the 2.5G-network because of the small amount of data needed to be sent between users.

6.1.3 Tibia Micro Edition

Tibia Micro Edition, also known as *"Tibia: Land of Heroes"*, or *"TibiaME"*, is based on the online role playing game Tibia. TibiaME is the first Massively Multiplayer Online Role Playing Game (MMORPG) for mobile phones, released for Symbian/Series 60 phones in May 2003. A Java ME version was released September 2006 [8]. The game can run on any mobile phone with Java MIDP 2.0 support, an Internet connection, and a color screen with a resolution of at least 128 × 128 pixels. A screenshot from TibiaME is shown in Figure 6.3.

In TibiaME, players explore the mysterious land of Tibia along with other players while fighting evil creatures and solving riddles to find treasures. By defeating monsters, the player gains experience points and grows in strength and power, which is helpful to defeat even more powerful monsters. Monsters also drop items when defeated, which the player can pick up and use. Players can also chat and exchange items with each other as well as fight other players on specific battle arenas. Player movement and interaction are real-time, whereas the fighting is turn-based.

The game consists of over 1000 different screens in which the players can fight more than 20 different creatures or find more than 80 different items. Items can also be sold and bought in shops placed around in the game world. Players can save their game and progress on the server at any time and continue whenever they want. When starting the game with a new player

Figure 6.3: Screenshot of Tibia Micro Edition (2006) from CipSoft

account, players can choose between playing as a warrior or as a wizard. These two character choices offer two different gameplays and tactics. The warrior emphasizes physical power and strength, whereas the wizard emphasizes magic and potions to defeat monsters and opponents.

Business Model

The game is available in two different versions, a free version and a gold version. The gold version adds more content and functionality to the game. The free version is available for download from the developer's website, while the gold version costs between $4 and $5. There is also a Premium character account subscription available that adds even more features. This subscription costs an additional $3 per month. Premium accounts have priority on servers and have access to exclusive areas. The game was launched in cooperation with Germany's largest mobile carrier, T-Mobile, in 2003. T-Mobile supports the game with download portals both in Germany and Austria. The game is upgraded at regular intervals and the upgrades include improved gameplay, extra features, additional islands and maps, and extra content like monsters and items. These updates are free for all versions, but some of them are only available for players with a premium account.

Network Technology

All players connect to a central game server, hence the game utilizes a client-server architecture. TibiaME is optimized for the GPRS and UMTS technologies. In an average hour of play, the player sends 400 kilobytes of data to the server. All the game servers are located in Germany. Because of this, the popularity is strongest in Germany, Austria, and Poland. However, the game can be be played anywhere in the world. Extra content and upgrades can be downloaded from servers by using WAP or by transferring them from a computer.

6.1.4 Summary

All of these games are highly advanced multiplayer mobile games with a similar form of business model. Two of them require users to subscribe to the game while the last one offers the game for free with extended content and features available for a fee. In addition to the subscription, the user cost is determined by how much the user send and receive data packets, i.e. users pay for each data packet sent or received.

Two of the games are categorized as Role-Playing Games (RPG), whereas the last is an action game. Because of these genre differences, the games have very different requirements to the degree of real-time data update. The action games require more frequent data update and lower latency than the rest, since player interactions are more turn-based in the RPGs. However, the network technologies used in these games (GPRS and UMTS) have high enough performance to deliver a satisfying and entertaining gameplay. The games' network emphasis is to ensure that the latency is kept low so the gameplay feels fluent.

The games attempt to provide a community outside of the game with leaderboards, player-to-player chatting, and downloadable content to ensure the popularity of the game. With such a community, players get an extra incentive to continue playing the game, and the game keeps generating income for the service providers. Downloadable content increases the longevity of the games and allows for problems or issues with previous game versions to be fixed.

These games show that multiplayer mobile game are more than feasible to develop and that current mobile network technology is capable of providing good enough network stability and speed for such games. However, not all the games are dependent on real-time data update. Games that demand such frequent data updates must use low-demanding user interaction methods, smart prediction methods, or suitable graphic updates so that the delays and slow data updates would not be noticeable for the player.

Part III

Own Contribution

Chapter 7

Prototype Game

As previously mentioned, the goal of this master thesis is twofold: on the one hand we are to develop a prototype game that tests the different mobile network's suitability for real-time multiplayer games, whereas on the other hand we are to perform general performance tests for the mobile networks related to mobile gaming. This chapter contains a short introduction to, and description of, the game prototype we will develop. In the final chapter of this part, we will describe our test modules and how they will be implemented.

7.1 BrickBlock

The concept and rules of the game, BrickBlock, is the same as in our depth project [29], therefore some of the following sections about the game is based on that report. Some aspects and areas are changed, rewritten, and extended to fit with the new network technology, game architecture, and game focus.

7.1.1 Game Concept

BrickBlock is a fairly simple multiplayer game where each player controls his brick around a two dimensional board. The goal of the game is pushing the other players into certain areas defined as traps. When a player's brick touches a trap area, he dies and receives a negative point. The winner of the game is the player that has died the fewest number of times, i.e. the player with the score closest to 0.

This concept opens for tactical play, as the players most likely will have to find other players to cooperate with in order to push and block the other players. In addition, these alliances will have to be temporary for one player to be a lone victor. Ambitious players most likely will jump from one alliance to another several times during the game to make sure he is always in the best position for the victory. In other words, BrickBlock will be a game characterized by its anarchy, chaos, and treachery, attributes that will make it an entertaining, unpredictable, and social game. Figure 7.1 shows an example of the gameplay with game objects, which are described in Section 7.1.3.

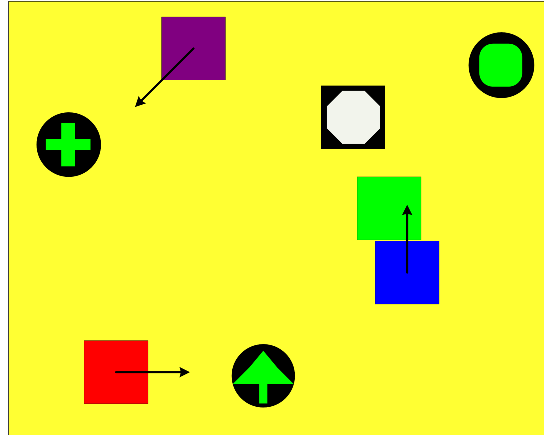

Figure 7.1: Conceptual model of the BrickBlock game

7.1.2 Game Rules

The goal of the game is to push other players into marked trap areas in the game map by controlling a brick. The players' bricks' speed, strength, and size are identical for everybody, except when a player picks up power ups. Power ups alter the properties of a player and changes the playing field, see Section 7.1.3. When a player's brick touches the trap area, the player dies and receives a negative point. The dead player is respawned at a random corner on the map after dying. The winner of the game is the player with the highest point value after a set time limit or when a player has died a set amount of times. These limits are set by the players before the game starts. The game can be played either in a free for all mode where every player is on their own, or in a team mode where the players are teamed up against each other. The game mode is also selected by the players before the game start. Assignment to a team is done automatically by the server, which distributes players evenly in two teams.

7.1.3 Game Objects

The objects in BrickBlock are displayed in Figure 7.2. In the game, each object has a base size of 10 $\times$ 10 pixels. Because of their limited size, the game objects have to be easy to interpret and understand. The power up objects uses the contrast between clear green and black so they can be easily separated from each other. The color of player bricks are generated automatically when a player logs in to the game. Each player has a unique color so the players quickly understand which brick belongs to which player. When team mode is enabled, the teams are represented with the team color in the top left corner of the player brick. This will allow players to understand which bricks are teammates and which are enemies.

The power ups featured in the game tweak the player's brick's abilities for a short amount of time when it is picked up. Picking up a power up involves navigating the brick to the power up icon on the map. The power ups emerge randomly on the map and are available for a set amount of time. The following power ups are implemented in the game:

Speed power up that gives the player increased speed, thus helping with avoiding other players or picking up other power ups.

Size power up that increases the size of the player's brick. This means that the player's brick

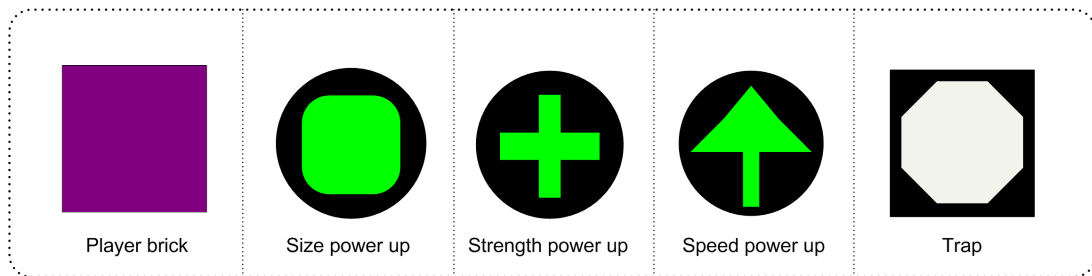

Figure 7.2: The game objects in BrickBlock

covers more of the map and therefore have greater possibilities to pick up more power ups or to be able to push the other players. It can also be seen as a negative power up since it will be easier to touch the trap area.

Strength power up that increases the player's strength, which means that the player will be able to push the other players more easily since they will be weaker and give less resistance.

7.2 Game Framework

While BrickBlock is a complete game in itself, it is very simple and offer little breathtaking functionality such as fancy graphics and intelligent Artificial Intelligence (AI). Our goal for the game is implementing a prototype for evaluating mobile networks, and not a "killer app". Because of this, such fancy functionality is neither required nor reasonable for BrickBlock. However, BrickBlock will contain much functionality that can be reused in other, more advanced games with small or no changes required. Examples of this is a login screen when starting an application, and a "lobby" screen that lists the players connected to a specified session. Simple in-game functionality such as basic collision detection and movement is also used in most games.

Because of this reasoning, the game architecture and implementation should be designed with close though to ease of modification and extension of the existing functionality. For example, if a developer should want to develop a car racing game based on the results of this master thesis, the pre-game functionality would likely be very much the same as for BrickBlock. The changes in game concept would lead to quite different game functionality. However, also here the properties of the player's objects (cars), collisions with game objects, and player movement will contain some of the functionality found in BrickBlock.

The functionality expected to be reused in other real-time multiplayer mobile games should therefore be extracted from the functionality specific for BrickBlock. This common functionality can then be placed in a framework that will give future game developers a kick-start in their game implementation. As explained in Section 4.3, this kind of framework is called an application framework. In this case, BrickBlock will then also be implemented by making use of the fundamental functionality found in the framework. Because of BrickBlock's simplicity, the necessary extension is likely to be small, and only contain functionality related to traps, power up objects, and players pushing each other.

Chapter 8

Server- vs. Client-side Calculations

When playing a multiplayer game over a network like the mobile networks discussed in Section 5.3, information sent between the participants will take a little time to reach the receivers. Because of this, synchronization of clients is a challenge, as it is hard (or impossible) for all clients to have complete control of the other clients' state at any time. Some calculation and prediction is therefore necessary in order to approximate a correct model of the current game state.

As previously explained, the first version of BrickBlock used a peer-to-peer architecture, where all the clients communicated directly with each other. Because of the equality of the clients in peer-to-peer networks, there was no natural organizing entity that could take command over the other participants. However, for some tasks, such an entity was needed. To compensate for this, we introduced the "game master", a role dedicated to the client starting the game.

With a server-client architecture like we have planned for the new version of the game, all information runs through the server before it is forwarded to the clients. Since the server is not equal to the clients, it can easily take on the role as game master where this is needed, and release the client of this responsibility. Since the server is involved in all communication, the information from one client reaches the server before the other clients are notified. In practice, the sum of information stored on the server will therefore in most cases be more accurate than on the clients. However, each client contains the most correct model of its own state.

Because of these varying degrees of accuracy, some of the calculations needed in the game are best suited for performing on the server, whereas other calculations are better handled on each client. Another factor influencing this decision is the previously mentioned difference between resources on a server and a client. A mobile phone is usually much weaker than a regular computer in terms of processing power.In addition, mobile phone has limited battery so power consumption has to be taken into account. Because of this, calculations that *can* be performed on both sides are in most cases best handled on the server. The following sections describe the most important aspects of BrickBlock where different calculations are needed, and whether these calculations should be performed on the server or on the client.

8.1 Collisions

Collisions is one of the most important aspects in nearly all computer games with moving objects. For shooter games, collision detection is needed to detect when the players shoot each other or run into each other. For classic games like Tetris, collision detection is needed to stop the bricks in the correct position. Even in games where collisions do not have immediately visible effects, like simple driving games, collision detection is needed for example to detect when the driving surface changes, i.e. when the car goes off track and onto grass.

In BrickBlock, the need for collision detection and handling is obvious. Without collision detection, pushing other players is impossible, and nothing will happen if the players move across a trap. This section discusses the different situations where collision calculations are needed, and whether these calculations are better handled on the server or on each client.

8.1.1 Collision Detection

A collision in BrickBlock occurs when a part of a player's brick touches another object or a wall. This can happen when a player moves his own brick into the other object or wall, or when he is pushed. There are four main causes for collisions: collision with walls, power up objects, traps, or other players. In the following, each of these causes are discussed, and the best approaches for detecting the collisions are found.

Collision With Walls

Wall collisions occur when a player's brick moves to a position where it is partly or completely located outside the game board. This happens either when the player tries to move to this position himself, or when another player pushes him to this position. The first case is quite simple to detect, as this only requires checking if the next move causes the brick to end in an illegal position. If so, the move is disallowed. Since this is such an easy case of collision detection, self-caused wall collisions should definitely be handled locally on each client. Figure 8.1 illustrates this situation.

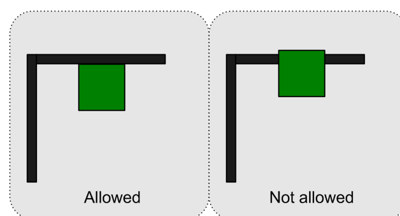

Figure 8.1: Self-caused wall collision

The other case of wall collision is a little more complex, as this involves interaction with another player. Several possible solutions are possible for this situation, the first of which equals the previous solution: If pushing a player results in that the other player is placed in an illegal position, the move is disallowed. The problem with this solution, however, is that the approximated position of the pushed player is not necessarily completely correct, because of the delay in information transmission. A move towards a wall may therefore be incorrectly disallowed, because a player that is not actually there is detected to be standing in the way.

Another solution is allowing such a move, and only checking the local player's position against the wall. In this case, a player may actually be pushed outside the wall, and this occurrence needs to be detected and corrected by either the pushed client or the server. In both cases, the simplest solution if such an event is discovered is sending a new position for the pushed player so that he is placed back in a valid position. If this is done on the server, the only client visibly affected by this is the player that pushed, as the pushed player will be moved to another position shorty after the push occurred. If it is done by the pushed player, all clients will be visibly affected, as they first receive a notification that a player has been pushed, and a corrected position shortly after. But the advantage of this last solution is that this detection is already mostly done through the calculation of self-caused wall collisions. A three-step illustration for this situation is shown in Figure 8.2.

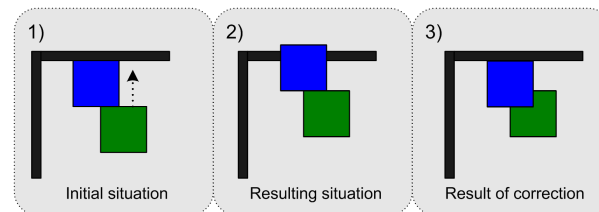

Figure 8.2: Externally caused wall collision

Unfortunately, as the figure shows, both the solutions involving correction when a collision is detected is likely to result in players being placed on top of each other when the pushed player is returned to a legal position. Because of this, we accept the mentioned drawback of the first solution and let wall collisions be detected by the pushing player. The result of this solution is that step 2 and 3 of Figure 8.2 are detected and stopped before they are executed.

Collision With Power Up Objects

Another type of collision detection is collisions with power up objects. When such an event occurs, the power up needs to be removed from the game board, and the player's attributes needs to be updated on all other participants. Like wall collisions, there are several solutions for this kind of collision detection, all of which have both advantages and disadvantages.

The simplest solution for detecting power up collisions is making use of the support for sprite collision found in the `javax.lcdui.microedition.Sprite` class in MIDP 2.0. This can be performed only by the local client, or it can be performed by all the clients each time a player moves. However, performing a collision detection each time a player position is received on all clients requires quite an amount of processing. As explained in Section 5.2, this is not desirable for a game designed for mobile phones, and should be avoided when possible. Furthermore, since information from one client takes some time to reach another, the player may experience the power up as available for some time after another player has picked it up. This may lead to several players picking up the same power up object.

Another possibility is comparing the player's position with the position of the power up objects on the server whenever a position update is received. A server is normally far more powerful in terms of resources than a client, but this solution leads to visible delay for the players, as the collision with the power up will not be registered before a little after the actual collision. The game should offer feedback to the player when colliding with a game object, such as vibration or flashing lights. If the collision detection is performed on the server, this feedback is likely to appear too late.

Hence, there appears to be a choice between sparing the client of these calculations and ensure that only one player can pick up a power up, or introducing a lag that can be avoided. However, the two solutions can be combined into a solution that both ensures only one player picking up a power up, as well as immediate feedback to the player. In this solution, the collision detection is performed locally on the client, as in the first solution, and the phone flashes and/or vibrates if a collision is detected. However, instead of immediately increasing the player's attributes, a notification that the player has collided with the power up is transmitted to the server. The server then checks if the power up has been picked up by any other players. If not, the server notifies all connected players that player X has picked up a power up object, and has increased one of his attributes. All clients must then remove the power up object from the game board once they receive the notification.

This solution still contain the problem with a lot of collision calculation on the client. However, in this case, the extra load for the client is worth the cost, because of the increased immediateness of the game. Therefore, the detection of power up collisions is performed locally on the player's client when he moves. When a power up collision is detected, a notification is sent to the server, and if the pick up is approved, the notification is forwarded to all connected players.

Collision With Trap

A trap collision occurs when a player collides with the trap object on the game board. This will usually happen when the player is pushed by another player into the trap, but it can also happen if the player is unlucky and moves himself into the trap. Both of these collisions equal the power up collisions discussed in the previous section, and is best handled by using the built-in support for collision detection in MIDP 2.0. Trap collisions and power up collisions are therefore detected equally and at the same time on the local client.

The problem with several players colliding with the trap at (close to) the same time does not apply to trap collisions as with power up objects. There is no rule against several players dying at the same time. However, when a player dies, he needs to be moved to an unoccupied corner on the game table. This involves traversing the player list and comparing the players' positions to the possible new position of the player. In itself, this operation is much like the collision detection already performed on the client. However, if several players die at the same time, all of these players need to be moved to an available corner. Because of the network latency, the new positions may be generated, and the players moved to the corner, before the other player's new positions are received. Hence, two or more players may be placed in the same corner if the resurrection position is generated on the clients.

Because of this problem, collisions with traps are handled in the exact same way as collisions with power up objects. If a player collides with the trap, his phone flashes and/or vibrates, and a collision notification is sent to the server. The server then generates the player's resurrection position, as well as the player's new score, and transmits this information to all players.

Collision With Other Players

Like collisions with power up objects and traps, collisions with other players are quite simple to detect using `Sprite` objects. However, power up objects and traps have constant positions and does not continuously move around on the game board like players do. As mentioned, it is impossible to have a completely correct overview of exactly where all the players in the game are at all times. This makes player collisions harder to detect correctly than collisions with other game objects, and even more difficult to handle in a satisfying way.

The simplest case of collision detection and handling between two players is when one of the players is standing still while the other is pushing. In this case, the collision detection is similar to game objects. The position of the pushed player can then simply be updated by letting the pushing player send a message that says that the player has been pushed to a new position.

But when both players move at the same time, the situation is more complex because of the network delay. This may result in three different situations. Figure 8.3 illustrate these situations for collisions between two players, but the same is true if three or more players collide.

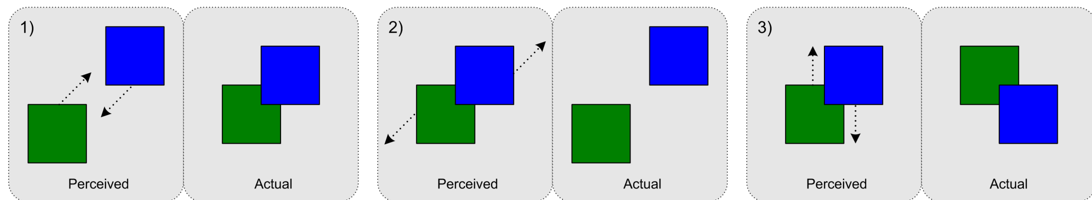

Figure 8.3: Player collisions with simultaneous movement

The left image of each case shows a possible representation of the player positions, whereas the right image shows the actual positions of the players. The three situations illustrated in the figure can arise when:

1. An existing collision is not detected because both players have moved into the same area, but the position of at least one player has not yet been received.
2. A non-existing collision is detected because both players who were in the same area have moved away, but the position of at least one player has not yet been received.
3. An existing collision is detected, but it is not completely correct since the position of at least one player has not yet been received.

The first of these cases may result in two players occupying the same board position for a short period of time, until the new position has been received and the collision is detected. However, this is not a very problematic issue, as the only time this happens is when the players touch very briefly, and does not try to push each other. The second case is the exact opposite of the first, and may in some situations be more problematic. The consequence of this case can be that a player is pushed even though he has actually managed to get away from the pushing player. If this happens close to the trap, the player may receive a negative point that he was not supposed to have. However, like in the first case, the correction will occur fast enough that we do not judge this latency to be a critical issue.

For the last of the three situations, there is no consequence for how the players experience the game. A collision is a collision, and whether this collision occurs at the edge of or at the center of the brick, the result is the same. A collision has occurred, and the strongest brick moves the other in the strongest's movement direction.

As previously mentioned, the server contains the most accurate approximation of the game state in sum, but each client contain the most accurate representation of its own state. This means that a collision that is detected on the server is more likely to be correct than one detected on the client. On the other hand, this solution introduces a visible latency to the game. The player will see that he collides with another player, but the effect of this collision will not register until the server has received the player's new position, detected the collision, and returned a collision notification. In other words, the player will experience that he is

moving a bit over the other player before the collision registers and the bricks start pushing each other. Because of this, and since the consequences of a little inaccurate collision detection are not too critical, we have decided to let each client be responsible for detecting collisions with other players.

Handling Player Collisions

When collisions between players are detected, these detections have to be handled so that the correct actions are taken. In BrickBlock, the results of such collisions are change of speed and movement direction for at least one of the players. In such an event, several factors need to be calculated. First, the strength ratio between the players involved in the collision needs to be calculated. If one of the players is stronger than the other, the strongest player will be able to push the other in the strongest's movement direction. How much the player can be pushed depends on the strength ratio between the players, as well as the movement speed of the strongest player. If the strongest player is 50% stronger than the weakest, and the speed of the strongest player is 2, the weakest player will be pushed with a speed of $(0.5 \times 2 =)$ 1. Since the players are pushing each other, the contact will be maintained, and the strongest player will also move with a speed of 1.

Like the other elements discussed in this section, collision handling may also be handled both server and client side. While the server has the advantage of plentiful processing powers, performing calculations on the client often leads to a more responsive game from the player's point of view.

In the case of player collisions and force movements, collision handling on the server profits from its more accurate world model compared to the pushing player, when it comes to calculating the new position of the pushed player. When the server is notified that a collision has occurred, it is able to calculate the new positions of both the pushing and the pushed player with relatively accurate values. However, the problem of visible delay on the involved clients once again arises. Both players will be able to move forward for a short time while the server is waiting for the collision notification, and when the server transmits the new positions, the players will experience that they are moved backwards seemingly without reason. This situation is illustrated in Figure 8.4.

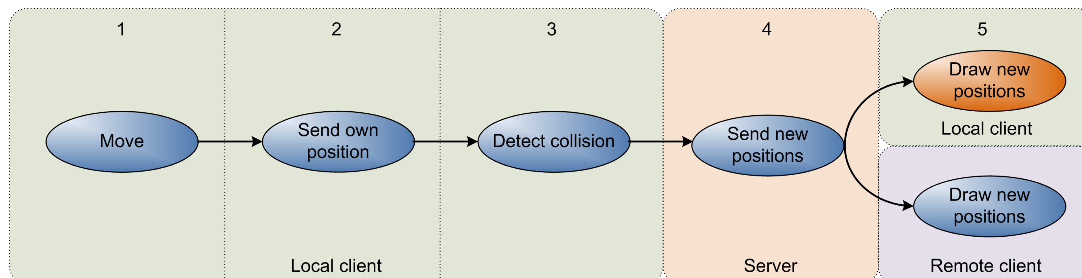

Figure 8.4: Server-side collision handling

The figure shows a step-by-step procedure of how calculations will be performed and messages transmitted when the server is responsible for handling player collisions. Where several boxes are placed over each other, the actions are performed in parallel. As the figure shows, the redrawing of positions happen first in step 5 on the local client. Two of these steps consist of transmission between server and client, and with a slow network, it is easy to understand that this solution involves significant delay for the players.

The other solution is letting the pushing player have responsibility for calculating the results of the collision. A step-by-step illustration of this solution is shown in Figure 8.5. Here, we see that the redrawing of the players happen already in step 3. Furthermore, no message transmission is necessary before the game board is updated. This will lead to a far more responsive game from the player's point of view.

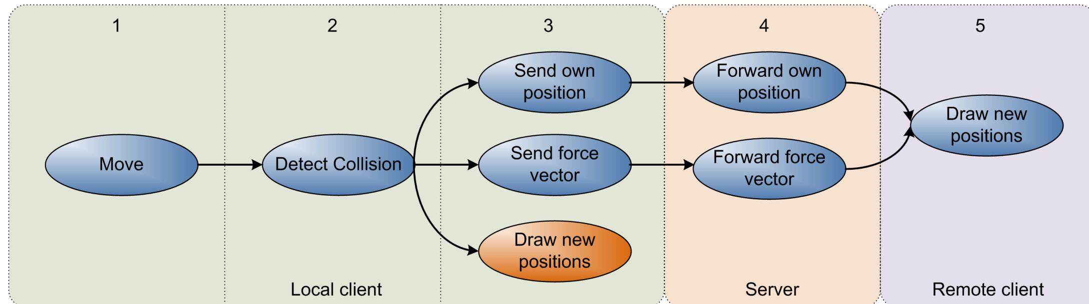

Figure 8.5: Client-side collision handling

As mentioned, there is a likely deviance between the other player's real position and its perceived location on the local client. Calculating the new position of the pushed player and transmitting this may lead to the same problem with seemingly unnatural position corrections. However, an improvement can be achieved by letting the pushing player transmit a movement vector instead of a static position. With this solution, the pushed player will not be reset to a previous state, but rather corrected with an amount corresponding to the strength ratio between the players and the speed of the pushing player. The procedure for detecting and handling player collisions on a client can then be as described in Listing 8.1.

Listing 8.1: Procedure for handling player collisions

```
1 Search for collisions
2 If a collision is detected
3         If I am stronger than the collidee
4                 Calculate the resulting movement vector for both players (will be equal)
5                 Transmit the collidee's movement vector to the server
6                 Calculate my new position
7                 Transmit my new position to the server
```

The movement vector solution could also be used on the server, and will reduce the problem of position corrections. But the problem with responsivity still remains. When a player collides with another player, he expects one of the players to be forced by the other player. As mentioned, with server-side collision handling, there will be a noticeable delay before this happens.

Because of the latency in the network, a player may experience to be pushed without there being contact between the players on his phone. Also, there may be situations where a push should occur, but does not. This is equal to the situations illustrated in Figure 8.3. With client-side collision handling, these situations will occur more often and with larger deviations than when performed on the server. However, we still feel that such situations will be a smaller source of irritation than the delay associated with server-side handling. As a consequence, collision handling is performed on each client when a collision is detected.

8.2 Game Control

In addition to collisions, the running of the game itself requires a number of calculations having to be performed throughout the game session. The state of the game is constantly changing, and events occur both because of player interaction and because of the game's inherent behavior. This section presents and evaluates the most significant of these events.

8.2.1 Power Ups

Power up objects in BrickBlock are generated with random intervals spanning from a minimal to a maximal value. To keep track of these intervals, the power ups should be generated only one place in the network. In the Bluetooth version of BrickBlock, the game initiator was appointed as game master, and had the responsibility for power up generation. However, using the same approach in a client-server network with relatively high latency may give the game master an advantage compared to the other players. He will see the power ups once they are generated, while the other players must wait for the notification before they are aware of the power up.

A better solution for the new architecture is letting the server handle power up generation. Some latency is still involved, and participants with slow connections may receive the notifications a little later than others. This was also the case in the earlier approach, and with this solution, none of the players have the game master advantage. In addition, none of the clients need to use their valuable resources for power up generation, but delegates this responsibility to the far more powerful server.

There are two possible scenarios that causes the removal a power up object from the game board. The first is when the power up times out without having been picked up by any of the players. This is quite similar to generation of power ups, and should be handled by the server for the same reasons.

The other times power up objects are removed are when a player picks up a power up. This event was discussed earlier in this chapter, and the conclusion was that the clients should themselves detect when they collide with a power up object. When such an event is detected, the client sends a notification to the server, requesting permission to activate the power up. If the request is approved by the server, this power up activation is forwarded to all players, along with the attribute increment provided by the power up object. The clients are then responsible for removing the power up in question from the game board.

When a power up has been activated, it remains active for the player for a set time interval. Detection of when a power up is deactivated is also a task that can be performed both server and client side. However, if this detection is performed client side, the exact same calculation has to be performed on every one of the clients. Of course, each player can be responsible for his own power ups and send a notification when a power up times out. Still, this check has to be run rather often, and will occupy more of the mobile phones limited resources.

If this check is performed on the server, it only has to be performed once each time the active power ups are checked. Since the server also has the most available resources, detection of timed out power ups should be performed on the server. Hence, the server runs through all the active power ups for all the players with set intervals, and if a power up deactivation is detected, a notification is sent to all players, who then set the affected player's attributes accordingly.

8.2.2 Game Settings

To give all the players a feeling of being equally involved in the game, it is important that all players have the same opportunity to change the settings of the game. Examples of such settings are how long a game should last, how many players are allowed in the game, or possible score limits. The changing of such a setting should naturally be performed on the local client, and the changes in settings are transmitted to the other players through the server.

However, the control of these settings can be implemented in various ways. In the Bluetooth version of BrickBlock, the game master was responsible for handling these settings, and detecting whenever a change in the game occurred (such as a time-out or reached score limit).

This solution is also possible to use for the server-client architecture, by letting the game initiator be game master. The problem with latency discussed in previous sections will once again be present. But for this kind of events, this latency is not critical. If a game finishes a little bit later on one phone than on another, the latency is likely to be so small that nothing of importance happens in the meantime.

Another client-based approach for this kind of events is letting all the clients be their own game master. All the clients have control over the different settings, and if a limit is reached, the game is simply closed locally. However, both of these client-based approaches require some background calculations continuously running in the background on the client. Even though these calculations are not very demanding, a server-based approach is not in any way worse, and in addition, frees the client from having to perform the calculations.

A server-based approach to this task requires the server to have a complete model of the game, such as player scores, number of players in the game, and time elapsed. Some of these elements are naturally stored on the server (such as players connected to the game), whereas others can be implemented with a minimal amount of effort. In this way, the server can continuously check the state of the game, and (close to) immediately send a "game over"-notification when the game should be ended. As mentioned, this takes some calculation load off the clients. Furthermore, like with power up objects, this approach reduces the small downside of delayed "game over"-notifications mentioned for the game master client-side approach.

8.2.3 Object Positioning

When starting a new game, all players joined in the game, as well as the trap, need to be positioned on the game board. For the game to be experienced as dynamic, and avoid some players always having an advantage over others, these positions should be different for each game, and randomly generated. Naturally, these random positions can not be fully generated on each of the clients, as this would lead to players and traps being positioned on different positions on the different clients.

Therefore, each object's position (trap or player) can only be generated on one place. None of the players have a closer relationship to the trap than others, so there is no reason why any of the clients should generate the trap position. Since the "game master responsibility" has been transfered to the server for the previous situations, it is natural that the server is also responsible for generating the trap position on game initiation. This trap position is then transmitted to all the clients so that everybody sees the trap in the same position.

For player positions, the situation is a little different. Each player has a close relationship to himself, and can therefore generate his own position. However, this may easily lead to two players being positioned on top of each other when the game starts. This can be avoided

by checking the position of all players and making sure that two players don not occupy the same position. Doing this on the client is difficult for two reasons. First, this is a demanding operation with a lot of players in the game. Second, and worse, it requires a predefined order for which player should get a position first, who is second and so on. If this is not done, the players would have very different models of each other's positions.

The simple solution to this is letting the server generate the player's positions. This only requires the server to loop through the players list and find an unoccupied position for each player. The initial position may then be transmitted as a position update, just as when a player has moved.

A third time a position is randomly generated is when a player dies. Unlike the initial position for the game, a player is positioned in one of the game's four corners when he dies. This is to avoid the unlucky situation of a player being positioned right next to the trap immediately after respawn. However, the problem of players being positioned on top of each other may again arise if this is not taken into consideration. This can be performed by both client and server, as it only requires finding an open position in one of the four corners, which is not a very demanding operation. With a lot of players, all four corners may be occupied, and another position nearby needs to be found. Since this task is quite similar to the generation of initial positions, we have decided to let the server have responsibility for determining respawn positions as well.

8.3 Summary

This chapter has discussed a number of situations where calculations on either the server or the client are needed. For some of these situations, this decision is easy to make, as one solution is clearly better than the others. However, other situations have a less obvious best solution. To get an overview of the decisions made in this chapter, each of the different situations are listed below, along with a description of our chosen solution.

Wall collisions are both detected and handled locally when the player moves himself into the wall. A move resulting in a wall collision is simply not allowed. The same applies when a player tries to perform a move that results in another player colliding with the wall; the move is not allowed.

Power up collisions are detected on the client, but handled on the server. When a client detects that the local player collides with a power up, it notifies the server. The server then checks if the power up is still available, and if so, notifies all connected clients. When the client then receives this notification, it must remove the power up object from the game board immediately.

Trap collisions are also detected by the client and handled by the server. When the client notifies the server that a trap collision has occurred, the server generates a resurrection position, and notifies all connected clients.

Player collisions are both detected and handled by the client. When a player collision is detected, the client calculates whether the local player is strong enough to push the other player. If so, a vector containing the push direction and speed is transmitted to all the other clients.

Generation of power ups is handled by the server. The server generates new power up objects at random intervals, and notifies all clients when a new power up object has been generated.

Generation of trap is performed by the server on game initiation. The position of the trap is then transmitted to all connected clients.

Generation of player positions is also performed by the server on game initiation, and when a player is respawn after he has died. When a player position is generated, the position is transmitted to the affected player immediately and to the other players with the next position update.

Detecting uncaught power ups is handled by the server. Each power up has a limited duration within which it must be picked up, or it disappears from the board. When a power up has not been picked up within its duration, the server notifies all clients that they must remove the power up from the game board.

Detecting inactive power ups applies to when a player's power up objects are no longer active. After a set amount of time, a picked up power up is deactivated. Detecting such events and notifying the connected clients are done by the server.

Handling settings is done continuously by the server. If a limit determined by a setting is reached, the server notifies all connected clients that the current game is over.

The common denominator for all of these choices is that we have tried find the solution that to the highest degree ensures an equally fair game for all players, independent of the network they use for the game session. Furthermore, we have tried to find solutions that reduce the latency in the network as much as possible, and ensure a fast and responsive game. Finally, we have tried to delegate as many of the calculations as possible to the server, because of the clients' limited resources. However, when such a delegation obviously and significantly increases the latency of the game, the clients handle the calculations themselves.

Chapter 9

Game Flow

The game flow is very important in a multiplayer mobile game. Players must experience the gameplay as fluent and responsive. This means that all on-screen action must be updated as fast as possible on every client and all player-actions must lead to direct execution in the game. To ensure real-time like data updates, the application should be developed with careful consideration of network issues. Solutions to such issues are difficult to develop, as well as almost impossible to be optimal, without sacrificing other parts. This is particularly true with the limited resources available on mobile phones.

Besides network specific solutions, other methods to ensure good game flow involves clever calculation algorithms and smart programming to camouflage the network latency. For example, by "simulating" data updates until an actual data update is received. This will improve the flow of the game from the player's point of view. However, the difference between the actual and the simulated situation can lead to incorrect situation interpretations by the players. This chapter discusses possible solutions for ensuring a good game flow in BrickBlock and similar games.

9.1 Movement Prediction

Movement prediction comes from the requirement of close to immediate player feedback and the issues with network latency [2]. Immediate player feedback is not so hard to solve; when a player presses a button on his phone, his controlled object reacts accordingly. Network latency, on the other hand, is far more complicated. Loss of data packets or slow data transfer caused by poor network conditions can lead to warping. This means that moving objects seems to jump from one location to another [2]. A fun gameplay with a good game flow requires accurate representation of player objects on all clients. To avoid warping and to accurately represent the player position at any time, the game client predicts the other players' movement, based on his previous movement. This leads to position changes happening earlier than by using the positions received from the server. Thus, the player movement will be perceived as more fluent, especially when a player has picked up a speed power up.

The movement prediction is performed by calculating a vector based on the two latest known positions of the object. These positions are stored on the client and are replaced whenever a new position is received. When a new position is received from the server, the new position and the previous position are used to calculate a the movement vector by subtracting the old coordinates

from the new and multiplying the result with the player object's speed property. This movement vector is then used to move the player object while waiting for the next position update from the server. This procedure for movement prediction is further described in Listing 9.1.

Listing 9.1: Procedure for predicting movement

```
Last received position update from player A: (x0, y0)
Receive position update for player A: (x1, y1)
MovementX = x1 - x0
MovementY = y1 - y0
If MovementX != 0
        MovementX = MovementX / Math.abs(MovementX) // Operate with 0's or 1's
If MovementY != 0
        MovementY = MovementY / Math.abs(MovementY)
Each time the game board is redrawn
        A's position = A's position + (A's speed)[MovementX, MovementY]
```

However, this method has its drawbacks. The predicted position may be wrong, for instance when a player changes his direction. This will lead to warping when a position update are received from the server, as illustrated in Figure 9.1.

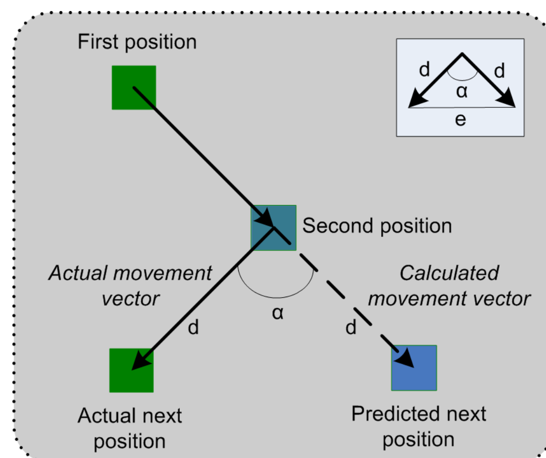

Figure 9.1: Movement prediction

The figure illustrates a situation where the player has performed a 90° turn. The more the player changes his movement direction, the greater the deviance between the predicted and the actual position will be. For example, if the player changes direction and moves back to where he came from (a 180° turn), the deviance between the predicted movement and the actual movement is greater than with a smaller direction change. A calculation of the prediction error, e, when the player has moved a distance of d after performing an α degree turn is shown in Listing 9.2.

Listing 9.2: Formula for prediction error

$c^2 = a^2 + b^2 - 2ab \times \cos AB$	*The cosine formula [47]*
$e^2 = d^2 + d^2 - 2 \times d \times d \times \cos \alpha$	*The formula applied to Figure 9.1*
$e = \sqrt{2d^2 - 2d^2 \cos \alpha}$	
$e = d\sqrt{2 - 2cos\alpha}$	

Using this formula, it is easy to see that the greater the direction change is ($\rightarrow 180°$), the greater the error (the warped distance) will be. This is further shown in Table 9.1, which

shows the prediction error for each of the possible α values. The values in this table are based on the worst case scenario, where the player turns immediately after a position update has been sent. Statistically, we can expect the average error to be half the values in the table.

α	0°	45°	90°	135°	180°	225°	270°	315°
e	0	$d\sqrt{(2-\sqrt{2})}$	$d\sqrt{2}$	$d\sqrt{(2+\sqrt{2})}$	$2d$	$d\sqrt{(2+\sqrt{2})}$	$d\sqrt{2}$	$d\sqrt{(2-\sqrt{2})}$

Table 9.1: Warp distances

In addition to warping, movement prediction may lead to problems calculating collisions with other players or with walls. If packet losses or disconnects occur, the player object will continue moving in the predicted direction. This may lead to an even greater position correction when the next position update is received. Finally, this kind of movement prediction demands a bit of processing power and client-side calculation. The more players in the game, the more predicted movement needs to be calculated. This will strain the mobile phone's resources and possibly slow down the game.

However, the gain of such a solution is invaluable for the flow of the game. Instead of the players jumping around on the screen at all times, which makes trying to push each other close to impossible, using movement prediction leads to a smooth and elegant game in most situations. Also, the problem with prediction errors is reduced with lower network latency, and participants in high-performance networks may not experience this problem at all. Because of this, movement prediction should be implemented in BrickBlock to improve the game flow.

9.2 Masking Delay

As mentioned in the introduction to this chapter, network latency can be masked by simulating position updates until an actual position update is received. The game "Pirates of the Caribbean" described in Section 6.1.1 is one example of a game making use of this technique. For example, when a ship in the game changes direction, the turning of the ship is animated on the client, and the position of the ship is not actually changed until a position update is received. However, for the players of the game, the movement appears to be realistic and real-time.

Other animations could also be used to hide the delay of the network. For example, instead of an implementation where the results of a collision, such as change of position, is immediately expected, an explosion can be simulated while the client waits for the update from the server. However, in our BrickBlock implementation, we have decided that the graphics shall be as simple as possible. This means that we will not use animations in our implementation, but the advantages of this method are definitely worth considering for other real-time multiplayer games.

Another way of masking delay is using interpolation [2]. This is a method that can be used in games where turning is made in a smooth movement instead of sudden changes of movement direction. An illustration of this method is provided in Figure 9.2. Here, the actual turn performed by the remote player is illustrated with the solid line and the green squares, whereas the calculated movement curve is shown with the stippled line and blue squares.

Because of the network latency, the turn is not detected immediately by the client, so that the predicted position differs some from the new received position. However, unlike the solution described in Section 9.1, the client does not immediately correct the error by warping the brick

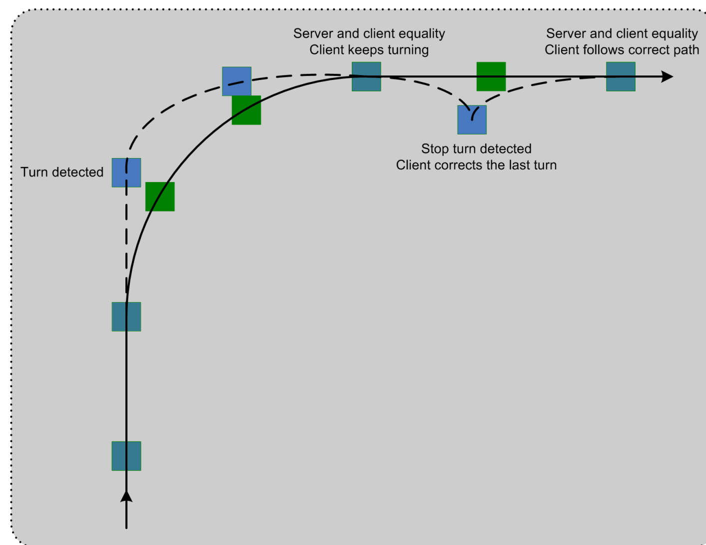

Figure 9.2: Interpolating smooth turning

into the correct position. Instead, it calculates the brick's position sometime in the future if it continues the same turn. Using this calculation, it moves the relevant brick along the calculated line. In this way, there will be a little deviance between the brick's drawn position and it's actual position, but the error will decrease the closer it gets to the predicted position. For the local player, this use of movement curves will lead to a game that feels much more smooth and elegant.

Unfortunately, this kind of masking also has its downsides. As the figure shows, the client does not necessarily detect that the remote player has stopped moving immediately, and may continue moving along the same curve too long. When this happens, the brick must be placed back on the correct line of movement without warping. A new curve must then be calculated based on a future location as shown in the figure.

In a game where the players move around a lot and turns frequently, interpolation may lead to the drawn positions of the players more often being wrong than not. Since the clients knowingly draw players in wrong positions, there will be inconsistencies between each client's model of the game board. In a game where the players do not affect each other by colliding, this problem can be ignored. In BrickBlock, where the players push each other when they collide, it leads to players colliding in seemingly incorrect places. A player following the solid line in Figure 9.2 may experience to be pushed by a player that is close, but does not actual collide because of these inconsistencies.

This is also a problem both when using the simple movement prediction from Section 9.1, and when movement prediction is not used in this project at all. Because of network latency, the states of the different clients can not be completely synchronized at all times, and erroneous collisions will be detected from time to another. A major advantage of this method is then that the deviance between the player's perceived position and his actual position is likely to be smaller than in the other two situations.

Like the other solutions to improve the game flow discussed in this chapter, interpolation also requires quite a bit of resources from the mobile phone. For each player, an approximated curve needs to be found, and with many connected players, this leads to a lot of calculations. Also,

use of smooth turning is not suitable for all games. For a car racing game, smooth turning leads to a realistic and life-like gameplay. For a game like BrickBlock, it is more likely to lead to frustration as it will be very hard to control the brick enough to hit and push other players. Because of this, we will not include smooth turning in BrickBlock, but implementing it in the game framework may be of great value for other developers using our game framework.

9.3 Message Bundles

Throughout the game, the server receives position updates from each connected client. These positions are sent in small position messages from each client to the server several times each second. They contain the positions of all clients and are used to update the objects' positions on the map. To keep the clients up-to-date on the game state, the server needs to forward the messages as soon as possible. The simplest solution to this task is simply having the server forward all position updates to all other players as soon as the position messages are received. However, trying to send many small position messages to many clients at once may lead to a congestion of messages, and an ever increasing queue of messages to be sent.

Another solution to this issue is bundling all position updates into one big message. Using this solution, the amount of information sent in each sent message is increased considerably, but the number of messages needed is decreased. In this way, the risk of message congestion is significantly reduced. Furthermore, all position updates will be received at close to the same time for all clients.

When using the message bundling solution, the server receives position updates in the same way as in the simple forwarding solution. These positions are then parsed into one message containing the id and the position of each client separated by a ';' character. This message is the large bundled message that is sent to all clients instead of sending several small messages. Figure 9.3 shows the difference between many small messages and one large bundled message. As the figures clearly shows, the number of messages sent are reduced with 1/3, even when only three clients are connected to the game. With more connected players, the number of messages will be further reduced.

A way of reducing the size of the bundled messages is removing the position of the receiving client, as he knows his own position. However, this requires generating different messages for each player, and complicates the process of generating the message. Considering that the reduction of message size is less than 1/n (where n is the number of connected clients), the drawback is greater than the benefit with this solution.

However, when using position bundles, it is important to be aware that the generation and parsing of the messages requires some additional time and resources compared to just forwarding the positions. The message generation on the server does not strain the resources remarkably, but the message parsing on the client will be more noticeable because of the mobile phones' limited resources. Since the message is so much larger and contains so much information, losing a message will also be more critical than compared to sending many small messages.

Still, withstanding from using message bundles may lead to unacceptable transmission delays because of the mentioned message congestion when many players are connected to the server. For a game with no absolute upper limit to the number of users, as we have planned, bundling of messages is therefore a method that definitely should be used.

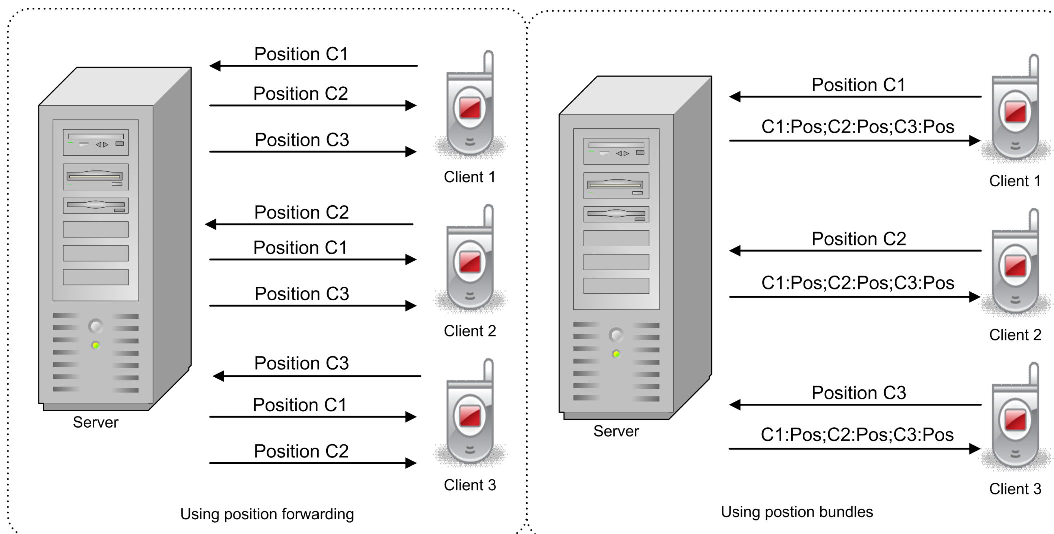

Figure 9.3: The two different methods for sending messages

9.4 Summary

This chapter discussed some of the techniques that can be used to mask and reduce the issues related to network latency. While not all of the methods are directly relevant for our BrickBlock game, they are still interesting for similar games with slightly different gameplays. Also, some of the techniques should definitely be used in the BrickBlock implementation. The following list contain a short summary of the different techniques, their advantages, and whether or not they are relevant for BrickBlock.

Movement prediction With movement prediction, the players will move around smoothly on the game board, even when position updates are received rarely. This method leads to somewhat strange behavior when a player performs a sharp turn, but its advantage is still great enough that we will implement it in BrickBlock.

Animation Using animations, direction changes can be masked by gradually turning the player object until it is placed in the correct direction. In BrickBlock, the player objects are represented by squares that never rotate. Because of this, this technique will not be applied to BrickBlock, but it is interesting for games that offer more advanced graphics.

Smooth turning Smooth turning makes use of interpolation to approximate the curve of a turn. This reduces the problem of warping. However, in BrickBlock, turns are performed immediately, with no smooth change of direction. Hence, smooth turning is also a technique that can be utilized in games with a slightly different gameplay.

Message bundling With message bundling, the server does not have to send a large number of small position updates, but rather sends larger message bundles regularly with a set interval. This reduces the risk of message congestion, as a lot less messages has to be sent from the server. Message bundling will be implemented in BrickBlock.

Chapter 10

Requirements

The requirements for a software system direct the functionality, the properties, and the architecture of that system. Requirements can be divided into functional requirements and non-functional requirements. Functional requirements are requirements that define the behavior and functionality of the system. These requirements describe a software system's internal workings and how calculations, technical details, data manipulation, and data processing are performed in the system. Functional requirements must be clear, unambiguous, and verifiable. Non-functional requirements, or quality attributes, support the functional requirements, and are requirements that specify the overall criteria for the application's operation. Non-functional requirements direct constraints on the design or implementation of the system and have a significant impact on the architecture and user satisfaction.

This chapter contains the requirements we have specified for our client and server applications. The functional requirements are derived by describing the functionality that should be implemented for the applications. At the end of these functionality descriptions, the formal functional requirements are presented in tables. By using this way of presenting the requirements, we ensure a comprehensible and coherent understanding of the application's behavior through the detailed description. In addition, keeping track of the requirements still waiting to be implemented is easy through consulting the requirements tables.

10.1 Client

The client application is developed to be run on mobile phones, and communicate with a server using existing mobile network technologies. This section contains an elaboration of the requirements we find necessary to ensure a stable, understandable, and entertaining game from the player's point of view.

As mentioned in previous chapters, the result of our earlier depth study was a BrickBlock game prototype based on a game framework (the peer2gaMe framework) [29]. Even though that version of BrickBlock was based on a P2P architecture, and used Bluetooth for its communication, a lot of the functionality specified for the first version of BrickBlock and the game framework still applies to this version. Therefore, parts of the requirement specification derived for the client in this section is based on our depth study.

10.1.1 Functional Requirements

Figure 10.1 shows a state chart containing the different states the client application may enter during a game session. In this chart, the state of the client is changed each time the client performs an action on the mobile phone. The labels annotated '<...>' represents a command available from the current state, whereas the labels annotated '[...]' represents some other action caused by either the local player, or another player connected to the game. For the rest of this section, the state chart is used as a basis for the description of the client application's functional requirements.

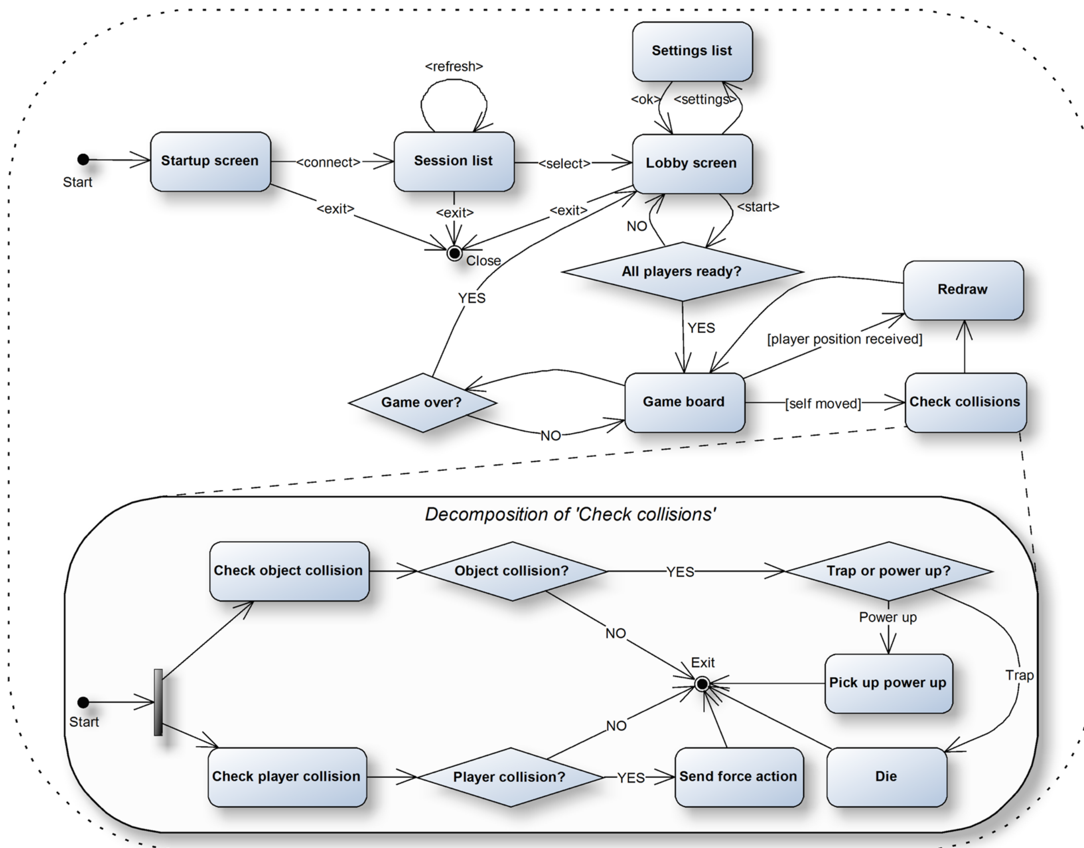

Figure 10.1: Client state chart

When playing BrickBlock on a mobile phone, the application is always in one of two main states, which offer quite different functionality and demands to player interaction and network communication. These two states are the pre-game and the in-game states. While the pre-game state is a relatively static and seldom changing state requiring little network activity, the in-game state needs constant updates through user input and network transmission. In this section, the two main client states and their functional behavior are described. A structured table of the functional requirements derived from these descriptions is provided at the end of the section, in Table 10.1.

Pre-game Requirements

Before actually playing a game, the client application needs to show a number of different screens where the player can enter information related to the game session. Examples of information needed is the player's nickname and the address of the server to connect to. In addition, screens for providing the player with server and session information is needed. Figure 10.2 shows the different screens available for the players before a game is started. In the following, the behavior for each of these different screens is specified.

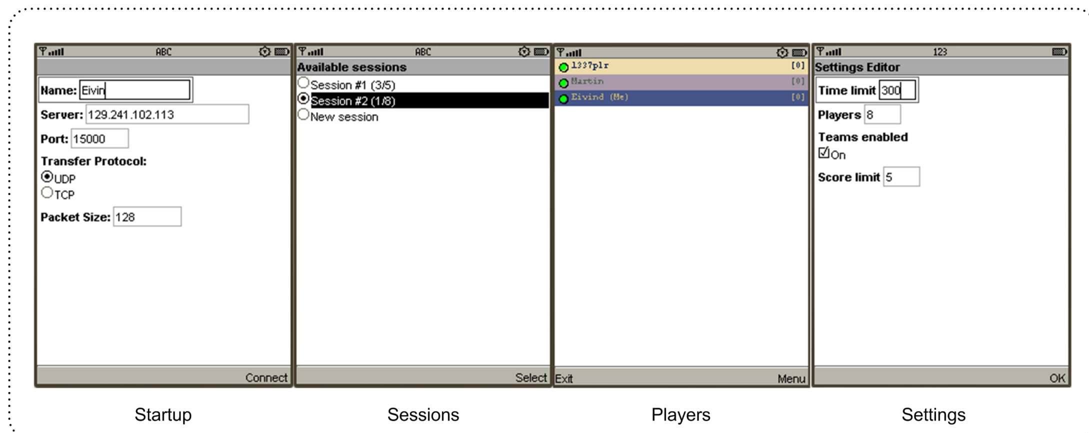

Figure 10.2: Pre-game screenshots

When starting BrickBlock on the mobile phone, the first thing the player should do is specify his name. In addition, to make the game server independent, the players should be able to select the server they want to connect to when starting the game. Therefore, a startup screen will be used where the player can write his name, as well as specify both the address of the server he wants to connect to and the communication protocol he wants to use.

When the server information is filled in, the player is ready to connect to the server. He will then receive a list of active sessions from the server. These sessions have a maximum number of players, and a name. The player can then connect to an active session if it is not full, or start a new session. If two players connect to the server at the same time, there will be two players that have not selected a session. To avoid these two having to start two separate sessions, it should be possible for the players to refresh the session list to see if there have been any changes.

After having connected to a session, the lobby view itself will be displayed on the screen. Here, a list of the connected players will be displayed, with player names, colors, and scores. Before a game has been started, all players will naturally have a score of 0, but between games, the result of the last game will be available by looking at the players' scores. In addition to the player list, the lobby should contain functionality for displaying and changing the settings. By allowing all players to change the settings at any time, the players will feel that they are part of the control of the game. This will hopefully introduce a feeling of "game ownership", and make the players want to play the game again and again.

BrickBlock supports teams, where one team is red, and the other is blue. If teams are enabled for the game, the players are placed on a random team. However, the teams will be balanced, so that one team never has more than one player more than the other. When teams are enabled, the lobby sorts the players according to their team, and shows the sum of points for the team.

To avoid starting a game before all the players are ready, the default status for all connected players is “not ready”. When a player feels that he is ready to start the game, he can change his status. This player status is displayed in the lobby as a red dot when they are not ready, and a green dot when they are ready. The game can not be started before all players are ready. After all players have signaled that they are ready, any player connected to the session can start a new game, hence changing the state of the client application to its in-game state.

In-game Requirements

When a new game has been started, the player is taken to the game screen. Here, each player is represented with a square block in the player’s color. In addition, the trap is represented as a hole. The positions of the players and the trap are received from the server.

During the game, power up objects are generated on the server and sent to the clients. When such a power up object is received, it must be drawn on the game board immediately.

Collisions with game objects, players and walls are discussed in detail in Chapter 8. Here, the conclusion is that all collisions should be handled locally on each client, and notifications sent to the server when collisions occur. To meet this requirement, the client needs to check for collisions each time the local player moves. If a collision is detected, an action is taken corresponding to the discussion in Chapter 8.

When moving around on the game board, the players’ bricks need to be drawn often enough that the game seems to run without lag. In other words, the Frames Per Second (FPS) rate should be high enough to ensure a good game flow. However, sending a new position update to the server every time the player’s brick is moved leads to an enormous amount of position messages to be sent. Because of this, the frequency of movement-packet transmissions should be lower than the FPS of the game [2]. For example, while the game board may be redrawn every 20 ms (an FPS of 50), it may be sufficient to send position updates every 100 ms.

Because of the latency in mobile networks and the low position transmission rate, the client is not likely to receive position updates from the other players more than a few times per second. Only updating the player positions when such messages are received is likely to lead to lagging and a poor game flow. In order to reduce this problem, the client must try to predict where the other players are likely to be positioned each time the game board is updated. To do this, one or more prediction algorithms are needed. Suitable algorithms for this purpose were discussed in Chapter 9, and one or more of these algorithms should be implemented in the game.

During the game, the participating players may want to view the score sheet without having to visit the player list in the lobby. In this way, the players may find out who are their hardest competitors for the win, or the weakest players in the game. An effective way of doing this is having a dial on the phone assigned for showing a transparent player list on the game board.

As discussed in Chapter 8, the server is responsible for detecting when a game is over due to a reached limit. When the player receives such a notification, he is immediately taken back to the lobby where the player list and each player’s points are shown. Thus, the clients state changes back to the pre-game state.

Summary

The previous section contained a description of the functionality in the client application of BrickBlock. From this functionality, a number of concrete functional requirements can be

derived. Table 10.1 contains the requirements needed to fulfill the desired BrickBlock functionality on the client side application. Here, the rightmost column shows whether this requirement is considered to be common for all real-time multiplayer mobile games, and should be included in the game framework (**FW**), or if it is BrickBlock specific. Functionality to be implemented in the framework is represented with a tick, whereas BrickBlock specific functionality is represented with a cross. This representation give an easily understandable way of separating BrickBlock specific functionality from common functionality in our implementation.

Table 10.1: Functional requirements for the client application

ID	Description	FW
C-FR1	Each player may write his own player name, of up to 10 characters and numbers (including space)	✔
C-FR2	Each player may select his own player color.	✔
C-FR3	The server address and port can be selected when starting the application.	✔
C-FR4	The communication protocol can be selected when starting the application.	✔
C-FR5	A player may start a new session at any time.	✔
C-FR6	A player may join a session in progress if the session is not full.	✔
C-FR7	A player may refresh the session list.	✔
C-FR8	A player may view the session's settings at any time.	✔
C-FR9	Any player in a session can change the settings for the session.	✔
C-FR10	Any player in a session can start a new game if all the players are ready.	✔
C-FR11	All players connected to the session are listed in the lobby.	✔
C-FR12	If teams are enabled, the lobby sorts the players according to team.	✔
C-FR13	A green or red dot signals if a player is ready for a new game or not.	✔
C-FR14	A player may enter the lobby at any time during a game without leaving the game.	✔
C-FR15	A list of the participating players can be viewed by pressing the *FIRE* button on the phone.	✔
C-FR16	The client must check for collisions each time the local player moves.	✔
C-FR17	Collision with power up objects is handled by the client.	✘
C-FR18	Collision with trap is detected by the client.	✘
C-FR19	A player dies if his brick touches the trap.	✘
C-FR20	A player can only be pushed by an equally strong or stronger player.	✘
C-FR21	The game board is updated every 20 ms.	✔
C-FR22	The position is polled and sent every 100 ms.	✔
C-FR23	A prediction algorithm is used to approximate each player's position between updates.	✔
C-FR24	Player positions are only sent while the player is moving.	✔
C-FR25	When the game is over, the player is taken back to the lobby.	✔

10.1.2 Non-functional Requirements

The non-functional requirements we find to be the most important for this multiplayer mobile game are usability, performance, availability, and modifiability. Explanation of these terms as well as the reasoning behind choosing these are represented in the following sections.

Usability

Usability is concerned with how easy it is for the user to accomplish a desired task and to what extent the system provides user support and help [5]. With high usability the game will attract more players, be easier to sell, give better player experiences, and have higher entertaining value. All games created are meant to be entertaining for the users. Understanding the game rules and goals are essential for the entertainment value. The same applies to the game interaction with the user.

C-NR1 The client shall be understandable and easy to use within 2 minutes of play without explanation beyond that provided in the game rules.

C-NR2 The user interface shall be easy to comprehend and interact with.

C-NR3 The client shall give understandable feedback to the user's actions.

Performance

Performance is about how well and fast the system respond to occurring events like interrupts, messages, requests, and so on [5]. The better and faster the system responds, the higher performance the system has. Ensuring high performance will lead to the game to be more satisfactory to play since there will be less chance of lag, delays, or other annoying aspects that will lower the game experience.

C-NR4 The client shall send the player action to the server immediately.

C-NR5 The data sent from the client should be limited in size to ensure the information is received fast on the server, i.e. within 0.5 seconds.

Availability

Availability is concerned with system failure and its associated consequences [5]. System failure occurs when the system no longer delivers a service as specified. This means that high availability demands a stable system that can handle different situations. A stable and consistent game will be more entertaining to play. A faulty game that crashes on a regular basis will be annoying and tedious to play.

C-NR6 The client shall handle faults in a way that keeps the user unaware of the fault, as well as letting the game continue running.

Modifiability

Modifiability is about the cost of change [5] and is concerned about adding, deleting, or modifying the application. High modifiability means that the application should be easy and simple to further develop or improve, yet still function as specified. Improving and extending the game requires work on the client application. Functionality may be changed or added. Also, the entire gameplay can be changed to make a completely different game. The higher degree of modifiability the client application has, the easier this job becomes.

C-NR7 A developer shall be able to add content to the client or change existing content without side effects on the rest of the application.

C-NR8 The client shall be usable as a basis for more advanced gaming concepts.

C-NR9 The client must support the transport protocols TCP and UDP on different network technologies.

C-NR10 Variables for adjusting the operational speeds of the client, such as FPS and send interval, shall be stored in a common class.

10.2 Server

Because of the first version of BrickBlock utilized a P2P architecture, that version did not need a separate server. Instead all the needed functionality was contained in the client application. Our new version of BrickBlock use a client-server architecture, and a server therefore needs to be implemented. This section contains the functional and non-functional requirements for such a server. As with the requirements for the client application, the formal requirements for the server are derived from a description of the server's desired functionality and behavior.

10.2.1 Functional Requirements

Figure 10.3 shows the states the server application may enter when hosting a BrickBlock game. This state chart is similar to that of the client shown in Figure 10.1, except that there are no labels marked <...>. This is because the "lobby state" of the server only depends on external events to change its state. There is no support for local user interaction on the server. Also, the server does not contain a termination point. The reason for this is that the server is not supposed to be shut down. To close the server, the local user has to exit the server application manually, which can be done from any of the states shown in the figure. As for the client's functionality, the server's functionality is described in the remainder of this section, based on Figure 10.3.

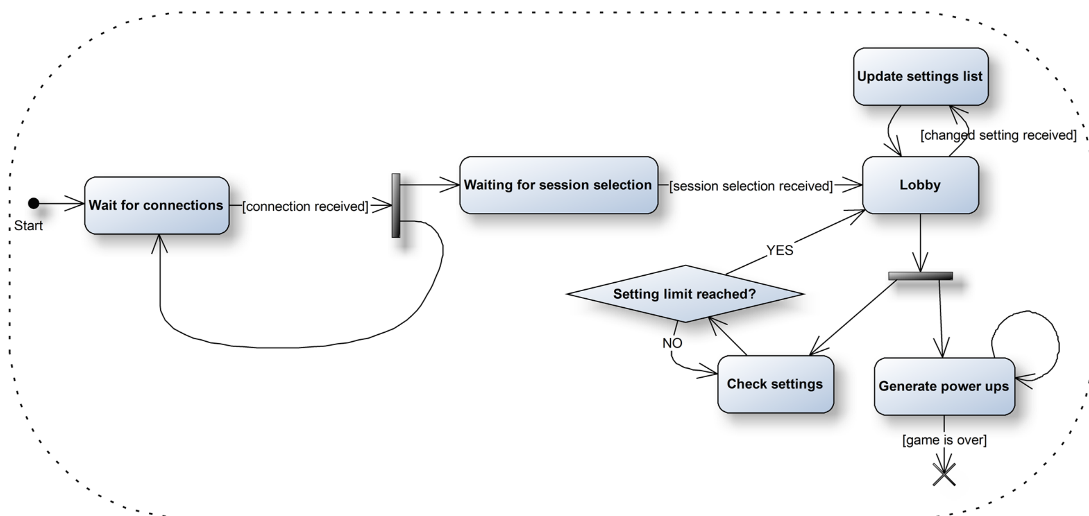

Figure 10.3: Server state chart

In this section, the functional behavior of the server is presented through a textual description, corresponding to the client's functional description in Section 10.1.1. Table 10.2 found at the end of this section presents the formal functional requirements derived from this description.

Like the client application, the server consists of two main states: the pre-game and the in-game state. The state change of the client application is very visible through its change of view to a game board. On the server, this change of state is not visible in the same way, as it will not contain a graphical representation of the currently running game. Still, the amount of calculations needing to be performed and the number of messages having to sent and received increases significantly when running a game. Because of this, separating these two states is also practical for the server application.

A server must be implemented, and contain the necessary functionality for running BrickBlock as intended. This server should be able to run on different platforms, to make deployment of the server as easy as possible. With such platform independence, the server can be run on any computer with an Internet connection. The server will therefore be implemented in Java SE, as this also simplifies reuse of similar functionality on both the server and client side applications.

Pre-game Requirements

As described in the client's functional requirements, the server should support multiple sessions, so that the players may find a session suitable for their wishes, or create a new session if wanted. The server therefore needs some way to represent the players connected to each session, as well as separate game threads keeping track of each session's current running game.

When a client disconnects from a session, independent of whether a game is in progress or not, it is important that this is detected by the server as soon as possible. This is important both to avoid spending resources on disconnected players and to give the connected players a precise feeling of the game. However, since the clients run on mobile phones, it is not always possible for the clients to signal that they leave the game before they disconnect. An example of this is when the phone runs out of battery and simply shuts down. To detect such events, the server should send regular requests that the clients need to respond to in order to tell the server that they are still alive. These requests must be sent both when the session is in its lobby state and when a game is running. When a client fails to respond to several subsequent requests, the client is disconnected and the remaining clients are notified of the disconnection.

One of the main factors in BrickBlock is the equality of the connected players. Any player may start a new game at any time, as long as all the players in the session are ready. The players may also change the game's settings at any time before a game is stared. To avoid several players changing a setting, or starting a new game at the same time, these commands need to be broadcasted immediately to all the session's players when detected by the server. When a start command is registered with the server, the session must change to its in-game state.

In-game Requirements

When the game initiates, the server is responsible for placing all game objects on the game board. The reasoning for this solution is found in Section 8.2.3. It is important that players are not placed on top of each other, so the server needs to run through the list of players connected to the session and generate a unique position for each player. In addition, the trap needs to be positioned on the game board. Here, the trap should never be placed too close to the walls, so that it can be reached from all directions.

When a player dies by moving (or being pushed) into the trap, he should be resurrected in one of the game board's four corners. Like with the position generation on game initiation, the

server needs to find an unoccupied position for the player. If all four corners are occupied, the server needs to find another position for the player, that is still as close to a corner as possible.

The power up objects used in BrickBlock are the server's responsibility. The current session's game thread should determine when it is time to add a new power up object to the game board, and generate this object's position. To make the game unpredictable, the time of generation and type of power up object should be random. In this way, none of the players will have an advantage over others by predicting when and where the next power up will appear.

In a game with many connected players, the server will receive a lot of position updates. If it should forward all of these updates upon receival, a lot of small position messages would have to be sent to each of the clients. This could easily lead to a congestion of packets waiting to be sent. As discussed in Section 9.3, a more effective approach to this problem is sending updated player positions in bundles with regular intervals. As long as this interval is not too long, the latency will be transparent to the players, and the problem with congestion can be avoided.

In addition to the position updates triggered by the players, the clients are also responsible for notifying the server whenever the local player picks up a power up object. To keep the state of the game as close to equal as possible for all connected players, this kind of information needs to be forwarded to all other clients as soon as possible. When such an event is detected by the server, it should therefore be broadcasted immediately.

Finally, the server has the responsibility for detecting when a game is over. This happens when one of the predefined exit states for the game is reached. Examples of this are when a player reaches the maximum (negative) score, or when the game has run for the set time limit. When such an exit state is reached, the server needs to notify all clients at once, and include a reason for why the game is stopped. Examples of such notifications are "Time is out!", or "<PlayerX> lost the game!". Other exit states, such as that all players have been disconnected, should also be detected and handled by the server as soon as possible. For the server to be able to do this, it needs to store the relevant information it receives from the clients and keep an up-to-date game model at all times. If all clients leave a session, the session should be closed to release resources.

When the current game has been stopped, and all clients have been notified, the server should close all running threads related to the game. If this is not done, an ever increasing number of running threads will occupy all the server's available resources and eventually cause the server to crash. Since the server should be able to host an unlimited number of games for unlimited time without having to be restarted, this must be avoided. After stopping the game threads and releasing the resources, the server must then enter a "waiting" state and wait for the players to start a new game.

Summary

In this section, the functionality of the server was described. Table 10.2 shows a list of functional requirements for the server derived from this description. Like the client's functional requirements, the rightmost column of the table shows whether the requirement in question should be a part of the game framework, or if it is BrickBlock specific.

10.2.2 Non-functional Requirements

The non-functional requirements we find to be the most important for this multiplayer mobile game server are availability, modifiability, and performance. Explanation of these terms as well

Table 10.2: Functional requirements for the server application

ID	Description	FW
S-FR1	The server is implemented in Java SE.	✔
S-FR2	The server can be run on any computer with an Internet connection.	✔
S-FR3	The server can run multiple simultaneous sessions.	✔
S-FR4	The server sends alive requests with regular intervals.	✔
S-FR5	A client failing to respond to alive requests is removed from the session.	✔
S-FR6	Changed settings are immediately forwarded to all clients.	✔
S-FR7	Start game commands are immediately forwarded to all clients.	✔
S-FR8	The server generates a trap position when a new game starts.	✖
S-FR9	The server generates player positions when a new game starts.	✔
S-FR10	The server handles generation of power up objects.	✖
S-FR11	The server generates power up objects with irregular intervals.	✖
S-FR12	The server notifies all participants when a player has picked up a power up object.	✖
S-FR13	When a player dies, he is placed (close to) one of the board's corners.	✖
S-FR14	Updated player positions are transmitted in batches with regular intervals.	✔
S-FR15	The server keeps track of the settings, and notifies all participants when the game is over.	✔
S-FR16	If all players disconnect from the session, the server closes the session.	✔
S-FR17	When a game is over, the server releases resources and stops threads related to that game.	✔

as the reasoning behind choosing these are represented in the following sections.

Availability

Availability is concerned with system failure and its associated consequences [5]. A system failure occurs when the system no longer delivers a service as specified. This means that high availability demands a stable system that can handle different situations. A stable server will be available for players a lot more often than an unstable server. For a multiplayer game to be attractive for players, the opportunity to play must always be present. When playing has commenced, the server must handle every situation in such a way that the players will perceive the game as flawless.

S-NR1 The players may connect to the game in progress at any time.

S-NR2 The players may disconnect from the game in progress at any time.

S-NR3 The server shall handle unexpected disconnections without affecting the game.

S-NR4 The server shall handle faults in way that keeps the user unaware of the fault, as well as letting the game continue running.

S-NR5 The server shall be operational 99% of the time, i.e. low mean time to repair.

Performance

Performance is about how well and fast the system responds to occurring events like interrupts, messages, requests, and so on [5]. The better and faster the system responds, the higher

performance the system has. Greater server performance means better gameplay and faster data updates for the players. A server without high performance will deliver an unsatisfactory game experience because calculation results can be wrong or completed too late. This will lead to low playability and displeased users, factors that every game developer wants to avoid.

S-NR6 The data sent from the server should be limited in size, while containing as much information as possible, to ensure fast data updates on the clients and low cost for the player.

S-NR7 The server's in-game generating and calculating tasks shall be completed correctly within 10 ms to ensure a fast and responsive server.

Modifiability

Modifiability is about the cost of change [5] and is concerned with adding, deleting, or modifying the application. High modifiability means that the application should be easy and simple to further develop or improve, yet still function as specified. With high modifiability, the server will be attractive for other multiplayer mobile game projects, since the gameplay can be altered or upgraded with less effort than if the modifiability was low.

S-NR8 A developer may add or change functionality without affecting the rest of the server.

S-NR9 The server must support the transport protocols TCP and UDP on different network technologies.

S-NR10 Values determining the operational speeds of the server, such as object generation and send intervals, shall be declared in an XML configuration file.

Chapter 11

Architecture

Bass, Clemets, and Kazman [5], defines the software architecture of a program or computing system as "*the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relations between them*".

In this chapter, the architecture for our BrickBlock application is described. As previously mentioned, the decisions made toward this architecture are based on the idea that the application should consist of a framework supporting ease of extension and a BrickBlock implementation based upon this framework. In addition, to support ease of modification and reuse of code, the server and client applications should be implemented in a symmetrical fashion. As far as possible, every class and method on one side should have a corresponding class or method on the other side. This eases the understanding of information flow in the system, and is an effective way to reuse code on both sides of the system. Figure 11.1 shows an architectural overview of the main components on the client and server side applications, and how information flows between these components. As the figure shows, both the client and the server consist of a three level architecture.

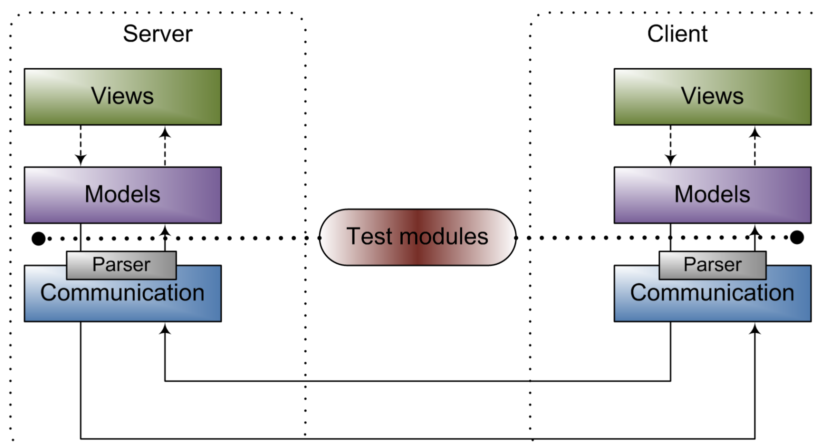

Figure 11.1: Architectural overview

In the bottom layer of the figure, we find the *Communication* modules. All communication between the server and the client run through these components. To ensure compatibility

between the communication modules, the server and client make use of common communication and message parsing interfaces. The parsers can easily be changed by using another parser implementation. This method effectively enables use of different message formats, such as plain text or XML. In principle, the communication modules only communicate with other communication modules and *Models*. However, when running tests, a *Test module* may be "plugged in between" the communication module and the models.

The *Test modules* are not a necessary part of the system in themselves, but can be used as an intermediate if a particular test should be run on either the server or the client. These modules also implement the communication interfaces, and can therefore be used as communication modules by the models. When using a test module, the models simply communicate through a test module, so that data can be stored and handled as needed before the data is forwarded to the communication module and send to the receiver. The test modules will not be discussed in this chapter, but a description of the test modules we will use are provided in Chapter 12.

The *Models* in the figure contain the information needed to represent the current state of the game. They also keep track of the messages needing to be sent, or being received. Both the server and the client make use of a number of different models for these purposes. This part of the architecture is where the mentioned symmetry can be utilized to its greatest potential. Both the server and the client need the representation of the game's current state to be as precise and correct as possible. Therefore, a lot of the information stored in the models will be equal at any given time. Because of this, the models found on the client side are likely to be needed on the server side, and vice versa. Hence, reuse of models is an efficient way to ensure client/server consistency, and at the same time reduce our implementation effort.

The top layer of the figure shows the *Views*. This is the part of our applications where we find the greatest differences between the server and client applications. While the server only needs a very simple view to show the players connected to each session, and each session's settings, the client views need to display all information relevant for the current game. In practice, this means that the server views only need support for showing the "lobby" information, whereas the client also needs to show the game board and update this immediately every time an event occurs. Furthermore, there are great differences between Graphical User Interface (GUI) implementations in Java SE (for the server) and Java ME (for the client). The result of these differences is that the reuse of code for the view components will be minimal.

The MVC Architectural Pattern

Our architecture shown in Figure 11.1 is based on the Model-View-Controller (MVC) design pattern. This pattern is a paradigm that provides for a separation of the features of graphical components, and support for MVC is well integrated in Java SE through the `javax.swing` package. Even though the integration of MVC is not quite as streamlined in Java ME, the advantages of MVC are great enough that the client implementation should also use an MVC approach.

The core of the MVC pattern is its functionality separation into *models*, *views*, and *controllers*. The *models* provide a storage mechanism for the information on which the application operates. The *views* use this models for rendering the information into a form suitable for interaction, typically user interface elements. Finally, the *controllers* process and respond to events, and if necessary, changes the models according to these events [15]. In this way, a single controller may detect an event, and update the necessary models as needed. All views using those models as basis for the information displayed can then change accordingly. In this way, cross-references

between modules can be reduced to a minimum, as all changes in the application run through the application's models.

As can be seen from the figure, our architecture contain the models and views as stated in the MVC view. When seeing the system as a whole, the *Communication modules* operate as controllers, as they detect messages from other participants (servers or clients), and forwards these messages to the models, who then change accordingly. All views using these models as basis then update themselves to display the models' contents correctly. By itself, each application (server and client) does not contain this sort of controller. Instead, the controllers are integrated in the views on local execution. The reason for this integration is that each separate view has its own well-defined functionality, and a change of one view does not directly affect the other views. Therefore, the views are responsible for detecting user input when they are visible, and notify the models that events have occurred. This kind of relaxed distinction between the view and controller is also referred to as the *Document-View* pattern [58]. The following two sections further describe the architecture of the client and the server application.

Client Architecture

Implementing a game framework to support easy development of future games similar to our BrickBlock prototype game requires identifying the functionality most likely to be reused later. In other words, the functionality deemed BrickBlock specific should be extracted from the framework. As mentioned in previous chapters, the lobby and communication functionality are very probable to be common for most games, whereas the functionality directly related to the gameplay is less likely to be reused. Still, some of the gameplay functionality is also likely to be reused, such as the underlying game board and basic movement. This version of BrickBlock is similar to the version developed in our depth study [29], and the high-level architecture is therefore also similar in most respects. Because of this, the contents of this section are based on the client architecture defined in our depth study.

Figure 11.2 illustrates the architecture of our client side application. The top three layers of the figure correspond directly to the layers of Figure 11.1, whereas the bottom layer represents the functionality found in Java ME. In the figure, the two modules written in italic illustrate the classes containing incomplete functionality, whereas the other two (and Java ME) contain all needed functionality.

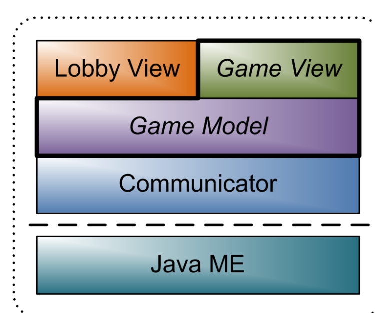

Figure 11.2: Client architecture

As shown in this figure, the *Communicator* receives message objects from the *Game Model* and sends these messages the server. Messages sent from the server are received in the *Communicator* and forwarded to the *Game Model*. From a networking point of view, the *Communicator* then acts as a controller, since it listens to and responds to network events. As explained,

having a separate layer for communication like this simplifies the process of changing the communication method.

The *Game Model* keeps a representation of the game's state at any time, as well as keeping track of messages that need to be sent, or that are received. The *View* layer is divided into two different parts; the *Lobby View* and the *Game View*. The *Lobby View* is responsible for providing the players with game information such as settings, participants, and scores when a game is not currently running. The *Game View* is responsible for the actual execution of the game, and needs to be able to display player avatars and game objects, as well as update its information whenever a game event occurs.

The bold lines surrounding the *Game View* and the *Game Model* represent the parts of the framework that need to be expanded when developing a game making use of the game framework. While the *Lobby View* and *Communicator* contain all the functionality needed, BrickBlock specific events and objects must be handled in the *Game Model* and *Game View* layers.

Server Architecture

Like the client architecture described in the previous section, the server architecture should also be based on the idea of a game framework that is easy to extend. However, since the server is not as closely related to the gameplay as the client, the use of abstract classes and unimplemented functionality is not as critical on the server as on the client. The server's most important functions are keeping track of connected clients, forwarding data, and sending position updates. This is functionality that most likely will be common for all real-time multiplayer mobile games.

Still, as concluded in Chapter 8, the server is responsible for some BrickBlock specific content during a BrickBlock game. This content is related to generation of BrickBlock objects like power ups and traps. An illustration of the high-level architecture for our server is shown in Figure 11.3. As can be seen, the layers of this figure correspond to those found on the left-hand side of Figure 11.1. Like the client architecture illustration, this figure also show the elements needing BrickBlock specific extension in italic font.

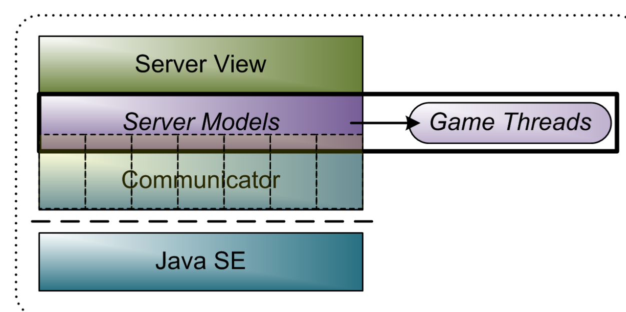

Figure 11.3: Server architecture

Analogous to the client's, the server's *Communicator* also receives messages from the *Server models* and sends these to the clients. Likewise, when the *Communicator* receives a message from a client, the message is forwarded to the *Server models*. However, unlike the client, the server contains many communication models. For each client connected to the server, a separate communication channel between the server and the client is needed. Therefore, each player model among the server models has a uniquely assigned *Communicator*, and all *model* ⇔

communicator communication use this relation. The relations are illustrated with the stapled boxes in the figure.

The *Server Models* correspond to the client's *Game model*, in that they keep a complete representation of the game state at any time. On the server, each running session containing players has a model representing that particular session, the session's current settings, and its players. When the players connected to that session decide that it is time to run a game, the session's model initializes its *Game Threads*, which are responsible for generating and handling game events.

Finally, in the top layer of the architecture, we find the *Server View*. This view shows the server's active sessions, and the session's connected players and active settings. As can be seen, the view is much like the *Lobby View* in the client architecture.

In the server application, the only modules needing extensions are the *Server Models*, and the *Game Threads* used by these models. This is illustrated by the bold lines in the figure. While these components need to be extended with BrickBlock specific content, the *Communicator*, *Server View*, and *Java SE* layers most likely already contain all functionality necessary.

11.1 Classes

Figure 11.4 shows a class diagram of the classes found in our client implementation, and the relationships between these classes. As the figure shows, the classes are divided into two main packages. The classes contained in the `framework` package are the classes found in the game framework, whereas the classes in the `brickblock` package show the implementation of the BrickBlock prototype game. For developers using the game framework as a basis for real-time multiplayer mobile game development, the classes in the `brickblock` packages should be exchanged with other game specific classes.

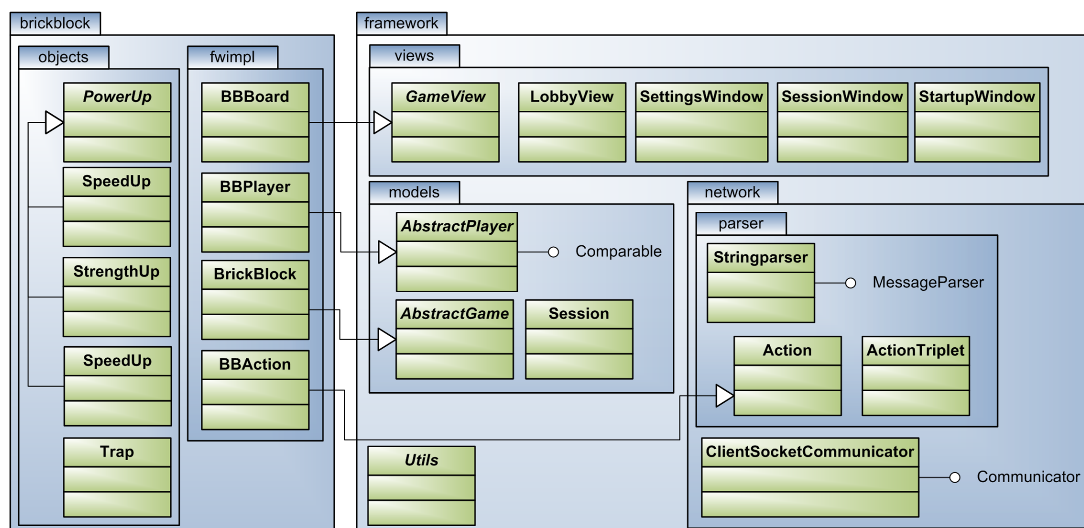

Figure 11.4: High-level client class diagram

In Figure 11.5, a similar class diagram is provided for the server implementation. Like the client classes, the server classes are also divided into a `framework` and a `brickblock` package.

As the diagram shows, a number of the classes and interfaces found in the client's class diagram are also present in the server implementation. This supports our wish for reuse of components, and is in accordance to our statements at the beginning of this chapter.

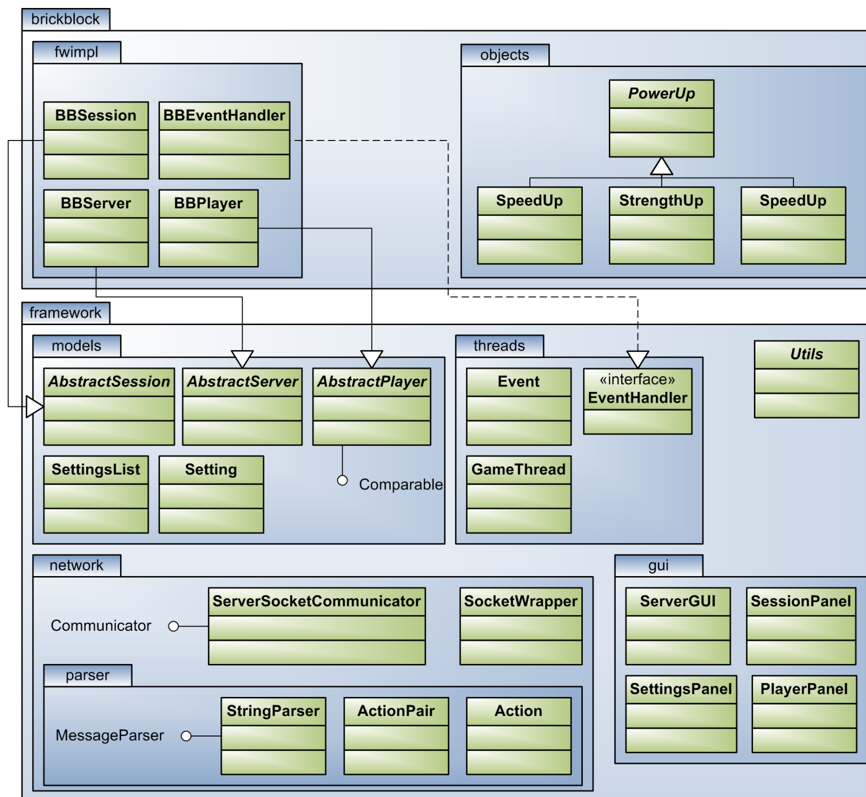

Figure 11.5: High-level server class diagram

The class diagrams shown in this section are high-level diagrams, showing the classes and packages used in our implementation. For ease of reading, methods and variables found in our classes are excluded from the diagram, as well as dependencies between the classes. For a more detailed overview and description of implementation specific details of our applications, consult the class diagrams found in Appendix D and the javadoc documentation.

The remainder of this chapter describes and explain the main components of our implementation, corresponding to the architecture shown in Figure 11.1. The most important of the classes found in the class diagrams are listed, and their purposes are explained.

11.2 Communication

The communication layer is the bottom layer of Figure 11.1. As explained, it is important that the server and the clients contain the same basis for communication to ensure that messages are sent and received correctly. For this purpose, we use a `Communicator` interface on both the server and the client that specify the methods that need to be implemented for the communication module. Using such an interface ensures that the communication modules on both sides support the same operations. Listing 11.1 show the contents of the `Communicator` interface.

Listing 11.1: The `Communicator` interface

```
public interface Communicator {
    public boolean connect(String[] addresses);
    public void disconnect(String[] addresses);
    public void sendMessage(Object message, boolean confirm);
    public void notifyAboutMessageReceived(Object message);
    public void close();
    public void searchForNodes();
}
```

The methods most important to notice in this interface are the `connect()`, `disconnect()`, `sendMessage()`, and `notifyAboutMessageReceived()` methods. The following list contain short descriptions of these four methods.

`connect(String[] addresses)` is called when connecting to one or more servers. In our implementation, only one server can be connected to at once, so that the list of addresses only will contain one address of the form <protocol>://<serverIP>:<serverPort>

`disconnect(String[] addresses)` is called when disconnecting from the server. In addition to sending a disconnect command to the server, this method should also clean up and release all resources occupied by the communication module.

`sendMessage(Object message, boolean confirm)` sends a message in the predefined message format. The message should be generated by a message parser, so that the format of the message is sure to be valid. The confirm flag can be used if confirmation of message receival is implemented. In our BrickBlock implementation, such confirmation is not available over UDP, but it can be supported in other implementations.

`notifyAboutMessageReceived(Object message)` is called whenever a message is received. It should notify its model that a message has been received and leave the handling of the message to the model. The format of the received message is ensured to be valid as long as a common parser is used on both the client and the server.

11.2.1 Protocols

As mentioned in Section 5.4, three relevant transport protocols are available in the Transport Layer of the Internet reference model, but we have chosen only to use two of these: TCP and UDP. Both of these protocols use server sockets for communication. On the server side, TCP uses a `Socket` for its end-to-end communication. Using this socket, the server is able to write information directly on a stream directed to the client. On the client side, a `SocketConnection` is used to read from this stream. When sending information from client to server, the same procedure is used. Because of TCP's connection-orientation, the connection only needs to be established once. After that, no specification of the receiver is needed.

For UDP communication, the server uses a `DatagramSocket` object, whereas the client uses a `UDPDatagramCommunication` object [27, 25]. Here, information is not put directly on a stream, but the messages are sent as data packets, and read into data packets by the receiving end. Since UDP is not connection-oriented, the address of the receiver has to be specified for each data packet that is sent.

In the BrickBlock implementation, the specification of which protocol to use is found in the `Server` class on the server side. Here, one can switch between either TCP or UDP as communication type. In the client application, the user can select the communication protocol to use when starting the application. UDP is selected as the default protocol. The reason for implementing such a choice is to enable testing of different protocols. A final version of

such a game meant for distribution to costumers should not contain this option. Instead, the choice should be done before distributing the game, based on tests resolving the most suitable communication protocol.

The `SocketWrapper` Class

The behavior of the game should be exactly the same no matter what communication protocol is selected. However, the mechanisms for this is quite different for TCP and UDP in Java SE. Because of this, we have created the `SocketWrapper` class on the server, which contain methods for writing and reading messages independent of the communication protocol in use. This class contain a `socket` object that is either a `Socket` or a `DatagramSocket` instance, depending on the protocol selected for the server.

Sending and receiving messages is then done by calling the send and receive methods found in the `SocketWrapper`. The differences of how this actually is implemented with the different protocols is thereby efficiently abstracted from the server's `ServerSocketCommunicator`.

In Java ME, communication over TCP and UDP both use implementors of `Connection` for sending and receiving messages. Because of this, no wrapper class is needed, since the `Connection` interface already specify the send and receive methods needed for communication [27].

11.2.2 Message Format

The communication between the client and the server needs to follow a well defined set of rules, so that all messages sent over the network are ensured to follow a valid format and contain legal values. For this purpose, a message parser should be used, which should be implemented similarly on both the client and the server. In our implementation, we use interfaces on both server and client to define the contents of such a parser. The contents of these interfaces are slightly different, but the parser's behavior should be consistent. Listing 11.2 shows the server's message parser, whereas Listing 11.3 shows the client's message parser.

Listing 11.2: The server's `MessageParser` interface

```
1 public interface MessageParser<T> {
2     public ActionPair parseMessage(T message);
3     public T createMessage(Action action, Object[][] values, Player sender);
4 }
```

Listing 11.3: The client's `MessageParser` interface

```
1 public interface MessageParser {
2     public ActionTriplet parseMessage(Object message) throws IOException;
3     public Object createMessage(Action action, Object[][] values);
4 }
```

As the listings show, there are two main differences between the server's and the client's message parsers. First, since the server is implemented using Java 6.0, generic data types are supported. This is shown by the *T*'s used in the interface specification and the method signatures. If one wants to create a message parser for String messages, the *T* can be exchanged with a `String` (or `XMLObject` for XML messages) in the interface implementation, and the parser will then only work for String objects. We had problems with the JWT's compatibility with Java version 5.0, and could therefore not use generics in the client implementation. Instead, the client interface specifies that the `parseMessage()` method should throw an `IOException` if an invalid message

object is received. The effect of this is, as with the generics approach, that only valid message objects are accepted, but the implementation is a little less elegant.

The other difference lies in the return type of the `parseMessage()` method, and the number of arguments for the `createMessage()` method. As further explained in the following sections, messages from client to server does not need a sender specification, since the server can easily detect the identity of the sender. However, when the server forwards a message to a client, the original sender needs to be specified in the message. As a consequence, the server needs to include the sender id when forwarding messages, and the client needs to extract this sender id.

The `ActionPair` and `ActionTriplet` objects are used as wrappers for the message contents. The `ActionPair` contains the action identifier as well as the values associated with the action. In the `ActionTriplet` wrapper, the sender of the message is also included.

The BrickBlock message format

The current implementation of BrickBlock only contain one `MessageParser` implementation: the `StringParser`. This parser only accepts incoming messages in a `String` format, and also only creates `String` messages. For the parsing of messages, a finite set of actions is specified. All these actions are defined in one single class on both the server and the client, so that adding or removing possible actions affect very few classes in a limited area. (In the client implementation, the BrickBlock specific actions are defined in a separate `BBAction` class, inheriting the default `Action` class.) Listing 11.4 shows an overview of how the parsing rules for the string messages are defined. The listing uses Extended Backus-Naur Form (EBNF) notation, which is explained in Appendix C. The rules listed in the listing apply for both server ⇒ client, and for client ⇒ server communication.

Listing 11.4: EBNF representation of the message format

```
<message>          ::= <actionstring> [ <playerstring> ] [ <valuelist> ] <endofmessage> ;
<actionstring>     ::= <action> <endofaction> ;
<action>           ::= ? An action defined in Table 11.1 or Table 11.2 ? ;
<endofaction>      ::= ":" ;
<playerstring>     ::= <playerid> <endofplayer> ;
<playerid>         ::= <intvalue> ;
<intvalue>         ::= <digit> { <digit> } ;
<digit>            ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
<endofplayer>      ::= "@" ;
<valuelist>        ::= <valuestring> { <endofvalue> <valuestring> } ;
<valuestring>      ::= <value> { <valueseparator> <value> } ;
<value>            ::= <intvalue> | <booleanvalue> | <stringvalue> ;
<booleanvalue>     ::= "true" | "false" ;
<stringvalue>      ::= ? A string of characters representing a value ? ;
<valueseparator>   ::= ","
<endofvalue>       ::= ";" ;
<endofmessage>     ::= "|" ;
```

When connecting a client to the server, a relation between the server and the client is created. This relation is maintained as long as the client is connected to the server. Each time the client sends a message to the server, the server knows exactly who has sent the message, and does not need a signature (sender id) included in the message to recognize the transmitter. Because of this, messages sent from a client to the server does not need the `<playerstring>` symbol specified in Listing 11.4. Figure 11.6 illustrates how a string message from the client to the server is composed.

For server ⇒ client communication, however, the situation is a little more complex. For some actions, like `score`, the receiving clients need to know for whom the action applies. In these cases, the `<playerstring>` symbol is used to specify the player affected by the action. In other

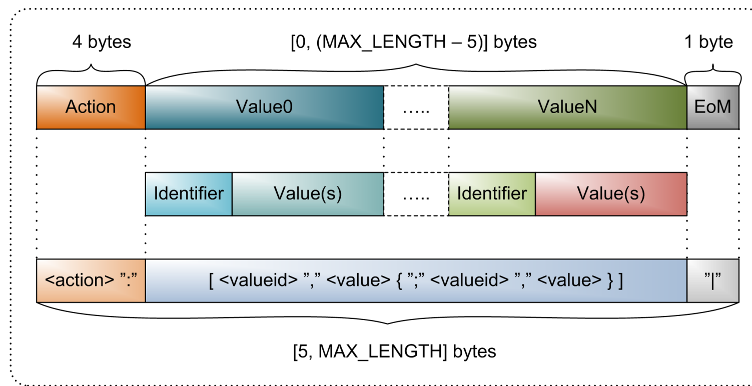

Figure 11.6: Format of the string messages sent from client to server

cases, like when a `trap_added` action is sent, no players are directly involved in the action, and no player ids are sent. In these cases, the server ⇒ client messages are equal to the client ⇒ server messages. Figure 11.7 illustrates the structure of a message from the server to a client, where the `Sender` field is optional.

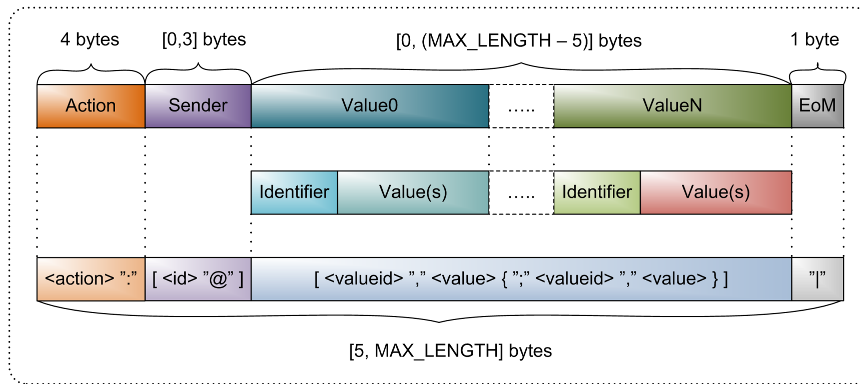

Figure 11.7: Format of the string messages sent from server to client

Actions

Table 11.1 shows a list of all actions used in the game framework. These actions and their contents are common for many games containing the framework's basic functionality. The table describes when the actions are used, and the contents of the string message following each action. In addition, the rightmost column in the table show the communication direction; if the action is sent from a client (C) to the server (S), from the server to a client, or in both directions. The actions in the table are listed in the order they are most likely to appear during a game session. Examples messages showing how the different actions are used can be found in Appendix D.

Table 11.2 is similar to Table 11.1, in that it lists actions used in BrickBlock. These actions are the ones specific for the gameplay of BrickBlock, and are not expected to be necessary for other games implementing the game framework. Like the previous table, Table 11.2 shows when

Table 11.1: List of actions used in the game framework.

Name	Value	When?	What?	Dir
PLAYER	PLR	A new player connects to the server, or requests a refresh of the session list.	Player information, such as name, color, team and other necessary information.[1]	Both
ALIVE_REQUEST	ARQ	A connection acknowledgment is requested or confirmed.	A unique request id.	Both.
SESSION_LIST	SSL	A new player connects to the server.	A list containing information about the currently active sessions.	S $\Rightarrow$ C
SESSION_SELECTED	SES	A player selects a session.	The id of the selected session, or a negative value if the player wants to create a new session.	C $\Rightarrow$ S
SETTING_LIST	STL	A new player connects to a session.	A list containing the selected session's settings.	S $\Rightarrow$ C
SETTING_CHANGED	SET	A player changes the value of a setting.	The name of the setting, and the new value.	Both
READY	RDY	A player changes his ready status.	The new ready status.[1]	Both
START	STA	A player starts a new game.	The screen resolution for the game, and the player's start position. (Nothing extra for the C $\Rightarrow$ S message.)	Both
SCORE	SCR	A player receives a (negative) point.	The player's new score and resurrection position.[1]	Both
POSITION	POS	One or more players have changed their positions.	For C $\Rightarrow$ S communication; the position of the local player. For S $\Rightarrow$ C communication; the id and position of all players that have moved.	Both.
GAME_OVER	GAM	A game is finished because a limit has been reached.	A description of why the game was ended.	S $\Rightarrow$ C
DISCONNECT	DIS	A player disconnects from the session.	Nothing extra.[1]	Both

[1] For server $\Rightarrow$ client transmission, the id of the affected player is included in the message.

each action is used, what the string message contains, and the direction in which the action is transmitted. Examples showing the usage of these actions can also be found in Appendix D.

Table 11.2: List of actions specific for BrickBlock.

Name	Value	When?	What?	Dir
TRAP_ADDED	TRA	The server adds a trap to the game board.	The position of the trap.	S ⇒ C
POWERUP_ADDED	PUA	The server adds a new power up object to the game board.	The kind of power up object, and the object's position.	S ⇒ C
POWERUP_REMOVED	PUR	A power up object is removed from the game board either by the server or a player (picked up).	The kind of power up object, and the object's position. When the message is caused by pick up, the player's id is included in the S ⇒ C messages.	Both
POWERUP_INACTIVE	PIN	A power up is no longer active for the player.	The power up object's attributes. The player's id is included in the S ⇒ C message.	Both
FORCE	FRC	A player pushes another player on the game board.	The ID and movement vector of the pushed player.	Both

11.3 Models

According to the MVC view, the models in an application contain the information needed to represent the current state of the application [15]. Both the client and the server applications have several models that represent different elements of the running game's current state. Also, as mentioned, we have tried to extract the functionality expected to be reused. This is used by creating abstract classes that can be extended with specific functionality when needed. Our decisions of what functionality to include in the game framework are based on the functional requirements derived in Section 10.1.1 and Section 10.2.1.

11.3.1 Client Models

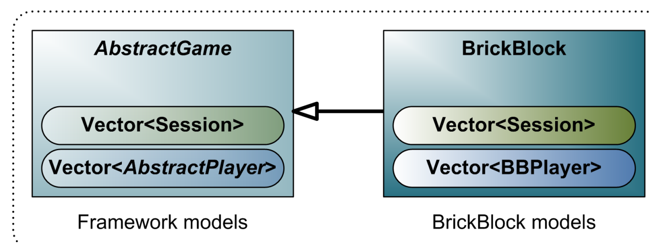

Figure 11.8: Client model representation

Figure 11.8 shows the models used for the client side of the application, and how they interact. The left box shows the classes belonging to the framework, whereas the right shows the models

used in the BrickBlock implementation of the framework. The *Vector<T>* annotation means that several instances of the model is used, for example by the `AbstractGame` model containing a list of `Player` models; one for each connected player. Even though we have not used generics in the client side implementation, this annotation is used as it clearly shows how the lists are meant to be used.

As the figure shows, the client application framework contains three main models. In these three models, the basic functionality for real-time multiplayer games is implemented. For the BrickBlock implementation (and other framework implementations), the functionality of the models is expanded by extending the classes, hence inheriting the functionality of the parent models. The following list describes the model classes found in the BrickBlock implementation. As mentioned, even though the list contains five classes, these classes only amounts to three separate models. The abstract classes are written in italic.

`AbstractModel` is the main model for the client, and also contains the `startApp()` method (equivalent to the `main()` method in a Java SE application). As the name suggests, this class is abstract and contains functionality found to be common for all games implementing the game framework.

`BrickBlock` is BrickBlock's implementation of the `AbstractModel`. In addition to the functionality found in the parent model, `BrickBlock` contains support for pushing other players and handling power up and trap objects.

`AbstractPlayer` is another abstract class containing common functionality for all games implementing the game framework. This model represents each of the players connected to the game, and their attributes, such as name, score, etc.

`BBPlayer` contains BrickBlock's implementation of the `AbstractPlayer` model. In addition to the functionality found in `AbstractPlayer`, this class contains support for the BrickBlock specific attribute "strength", as well as the player's currently active power up objects.

`Session` is a very simple model that only contain the sessions currently active on the server, their connected players, and their maximum number of players. This model is only used for the "Session list" screen, and because of its simplicity, it is not expected to need extension.

11.3.2 Server Models

Figure 11.9 shows a representation of the models used on the server side of the application. Similar to the client models' *Vector<T>* annotation, the *List<T>* annotation means that several instances of the specified model are contained in a `List`. Since the server can run several parallel sessions, there is a natural hierarchy of the server's models, as is clearly shown in the figure.

Like the client implementation, the server also contains a number of abstract classes, supporting the basic functionality found in the framework. By extending and implementing these classes, a new game with new gameplay can easily be created without having to reimplement the already existing functionality. In the figure, this is shown through the framework's abstract classes in the top layer of the figure, and the BrickBlock extension in the bottom layer. The following list contains descriptions of each of the classes found in the figure.

`AbstractServer` is the top-level model for the server application. This model contains a list of all the players connected to the server, as well as a list of the currently running

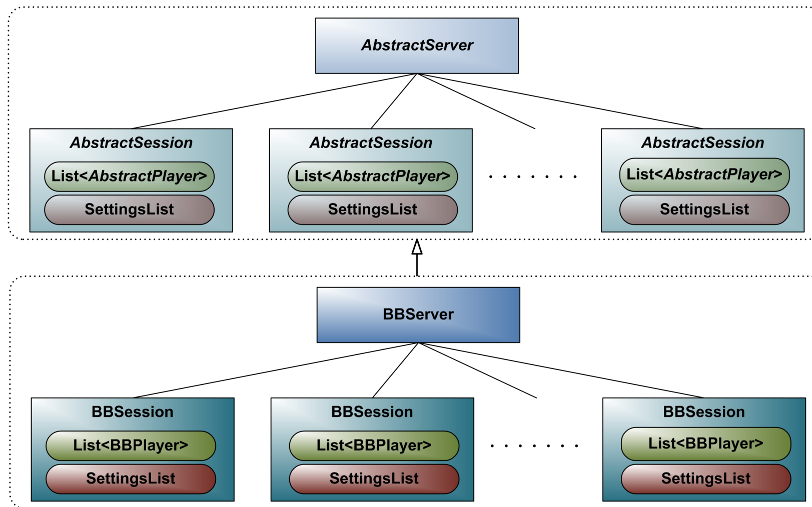

Figure 11.9: Server model representation

sessions. In addition, the server model is responsible for receiving notifications from the communication module when messages are received.

`BBServer` contains the BrickBlock extension of the `AbstractServer` class. This model contains very little extra implementation, and only keep track of the BrickBlock specific session and player models, as well as receive BrickBlock specific actions from the communication module.

`AbstractSession` is a model representing a session. On a running server with many players connected, the server is likely to contain several session models. Each of these session models contains a list of the players connected to the session, and keeps track of the settings applying to that particular session. Also, the session model is responsible for controlling the currently active game threads when running a game, and notifying the `EventHandler` when a message is received that is likely to cause a game event to occur.

`BBSession` is the BrickBlock version of the session model. Like the `BBServer` model, this model contain very little extra functionality extending the parent model's. In our implementation, this class merely serves as a specification of the type of player and event handler objects applying to the BrickBlock game server.

`AbstractPlayer` is very much equal to the `AbstractPlayer` model described for the client. The main difference is that each of the players represented on the server needs a uniquely assigned communication line to ensure that transmission and receival of messages are handled correctly. Because of this, each player model on the server contains a `Communicator` object used for client communication.

`BBPlayer` is, like in the client application, the BrickBlock implementation of `AbstractPlayer`. The only extra functionality found in this class is support for power up objects.

`SettingsList` is the only non-abstract model in the server implementation. This model only contains a list of the settings applying to each session, and is used by the session's `GameThread` to ensure the validity of the game's current state.

11.4 Views

The views in the MVC pattern are responsible for presenting the contents found in the models, and providing mechanisms allowing user input. For the client application, this means providing the user with functionality for connecting to, changing, and participating in game sessions. For the server, it simply implies some kind of representation of the server state, without user input.

11.4.1 Client Views

The client contain a number of views, where the two most important are the `GameView` and `LobbyView` views, as shown in Figure 11.2. In addition to these two views, three smaller window classes are used in the client application. These classes are used to allow user input to be entered and sent when needed. Which view currently displayed on the client is controlled by the `AbstractGame` model described in Section 11.3.1.

Like the model classes described in the previous section, some functionality in the view classes can also be extracted to the framework, whereas some functionality is BrickBlock specific. However, as the client's functional requirements in Section 10.1.1 explain, the only functionality specific for BrickBlock in our case applies when a game is running. Therefore, the only view class needing to be extended with game specific functionality is the `GameView` class.

`GameView` is the view that is responsible for showing the game board when running a game. This view uses the `AbstractGame` model for getting and displaying the current state of the game. Since the way the game board is displayed is very likely to vary from game to game, this view is declared abstract, and leaves a number of methods for its inheriting classes to implement. The only functionality specified in this class is the drawing of the underlying game board, and the player's sprites.

`BBBoard` is the BrickBlock implementation of the *`GameView`*. In addition to its inherited functionality, this view also contains methods for drawing BrickBlock game objects to the game board.

`LobbyView` is the "waiting screen" for players connected to a session that does not currently have a running game. This view contains functionality for showing the players connected to the session, their score, and their ready status, as well as commands for changing to the `SessionWindow` and `GameView` views.

`SettingsWindow` contains all the settings applying for the selected session, and their values. These settings are shown in editable components, allowing the users to change the settings' values.

`StartupWindow` is the first view that is displayed when starting the client application. This view contains text boxes for entering the wanted player name, as well as server data such as IP address and port number. As previously mentioned, the server and communication type fields should be removed in a final game distributed to costumers.

`SessionWindow` lists the currently active sessions on the specified server. For each session, the session's name, current player count, and maximum player count are displayed. From this view, the player may select and connect to an existing session, or start a new session.

As previously explained, our architecture does not contain separate controllers for handling local user input. Instead, this responsibility is included in each of the views, since the functionality of each view is well-defined and separate from the other views. Each of the views

therefore contains methods for detecting user input and notifying the models when such events are detected.

11.4.2 Server Views

The server views are very simple, and actually not needed at all to run a game server. However, to allow an easily readable representation of which players are connected to which sessions, and the settings that apply to these sessions, a simple GUI is still implemented for the server. The classes used for the server view are described in the following list.

`ServerGUI` is a `JFrame` extension containing one `SessionPanel` for each active session. A running server only have one `ServerGUI` object.

`SessionPanel` is a `JPanel` extension containing one `PlayerPanel` and one `SettingsPanel`. Each active session on the server have one corresponding `SessionPanel`.

`PlayerPanel` contains a representation of the players connected to the corresponding session. Each player is represented by his name, id, current score, and, if enabled, his associated team color.

`SettingsPanel` lists the settings currently applying to the corresponding session. Each setting is represented by the setting's name and value.

Unlike the client views, the server views does not contain controller functionality. The reason for this is simply that the server does not allow local user input, and that all externally caused changes in the server models come from connected clients. As a consequence, the only controllers on the server side are the communication modules.

11.5 Threads

For all sessions running on a server, the threads are organized as shown in Figure 11.10. Each session contains one *GameThread* and one *EventThread*, which together control the running of the game. In the game framework, these threads are called `GameThread` and `EventHandler`, respectively. BrickBlock's implementation of the `EventHandler` thread is the `BBEventHandler` class.

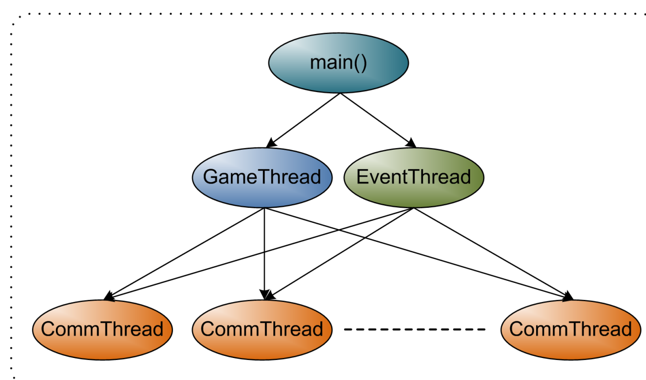

Figure 11.10: Server threads

The GameThread's main responsibility is keeping track of the players' positions at all times, and distributing the new positions to all the participants whenever one or more players have moved. To avoid having to send too much information too often, all updated positions are sent in a batch with a set interval (usually between 1 and 10 times per second). The most suitable value for this interval will be further tested and discussed in Chapter 13.

The EventThread is responsible for placing and removing objects, such as power ups and traps to the game board. Whenever the EventThread adds a new object to the board, all participants are notified immediately.

In addition to these two main threads, each connected participant has associated *communication threads*. The GameThread and the EventThread communicate directly with these communication threads and use them to send messages to and receive messages from the clients. Figure 11.11 further shows how these communications threads are organized, and how they interact with their associated clients.

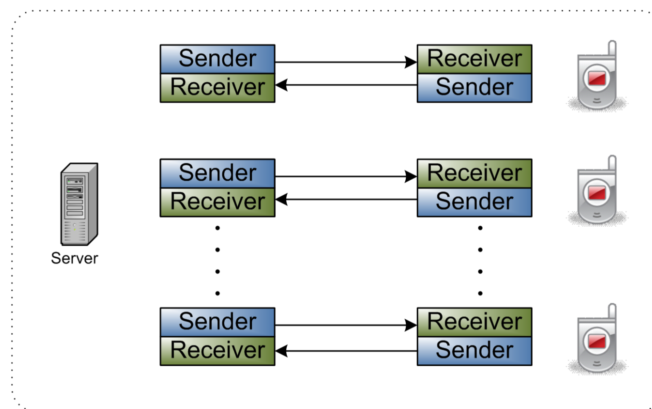

Figure 11.11: Communication threads

As the figure shows, the communication threads consist of one thread responsible for sending, and one thread responsible for receiving messages. Analogous, each client contain one sender and one receiver thread. These two threads run completely independent of each other. Ideally, the receiver threads would be continuously running to immediately detect and receive incoming messages. However, a continuously running thread occupies a lot of resources, so that a short sleep interval is used between each time the receiver thread checks its connection. The sender thread is paused whenever its send queue is emptied, and notified immediately when a message is ready to be sent.

Unlike the server, the client only contain exactly one thread for sending and one for receiving messages, as Figure 11.11 shows. Since each client communicates directly with one server, and delegates the responsibility for forwarding messages to the server, these two communication threads are sufficient for the client.

In addition to the two communication threads, the client also contains one thread responsible for the local calculation and one thread responsible for polling the player's current position at regular intervals. In the client implementation, these threads are called `LocalThread` and `PositionThread`, respectively, and are located in the `AbstractGame` class. These two threads run in parallel, but with different speeds. The calculation thread needs to perform its calculations, such as detecting collisions and updating the game board, each time the local player moves or an external event is received. Thus, this thread corresponds to the server's *EventThread*, with

its responsibilities determined in Chapter 8.

The client's `PositionThread` corresponds to the server's *GameThread*, in that its responsibility is sending the player's position with regular intervals. Each time this thread loops, it polls the player's current position and notifies the client's *SenderThread* that it shall send a position update to the server.

Chapter 12

Test Modules

The test modules in this project are used to perform performance tests of the different available mobile network technologies and transfer protocols. Each test case extends the *TestModule* class. This class contains the test setup variables, the test report functionality, and other useful functionality that is common for all the tests. Such common, useful functionality are for instance the methods used to convert from nanoseconds to seconds, the timer functionality placed in a nested class (`Timer`), and the `Sender` nested class that sends the packets. The setup variables determine the number of runs for a test, the number of intervals, and the number of packets sent in each interval. If the completed interval is the last interval, the run is completed. If the completed run is the last run, i.e. the number of determined runs is reached, the test is completed and a report of the results is created as a html-file. This report creation is delayed to ensure that the last packets are received and the measurements are completed.

12.1 Response Time

The response time test measures Round-Trip Time (RTT), the time a small packet uses from the server to a client and back to the server again. The test generates packets of only 4 bytes containing an id and a separator character. The number of packets generated depends on the number of intervals and the number of packets sent in each interval. These packets are sent with a delay of a set amount of milliseconds, which increases with each interval. The test calculates the time values, extracting the highest and lowest times, and calculates the average for the rest. These values can be used to determine the amount of milliseconds an interval's delay should increase with and the size of the start delay. The delay increases will show with what interval between sent messages the optimal performance, i.e. the lowest RTT, can be achieved.

Figure 12.1 shows how the ping test is performed, in this case by sending one packet in n intervals. With more packets per interval, the time between sends is not increased before all packets in that interval has been sent. The send packet is represented as an orange square with a length of l. The time the packet uses from the server to the client and back is denoted as t_i, where the i represents the packet's number. Finally, the send intervals are denoted as multiples of ΔI.

The purpose of the response time test is to find the send interval that gives the shortest RTT. With an optimal send interval, the communication between server and client will have low

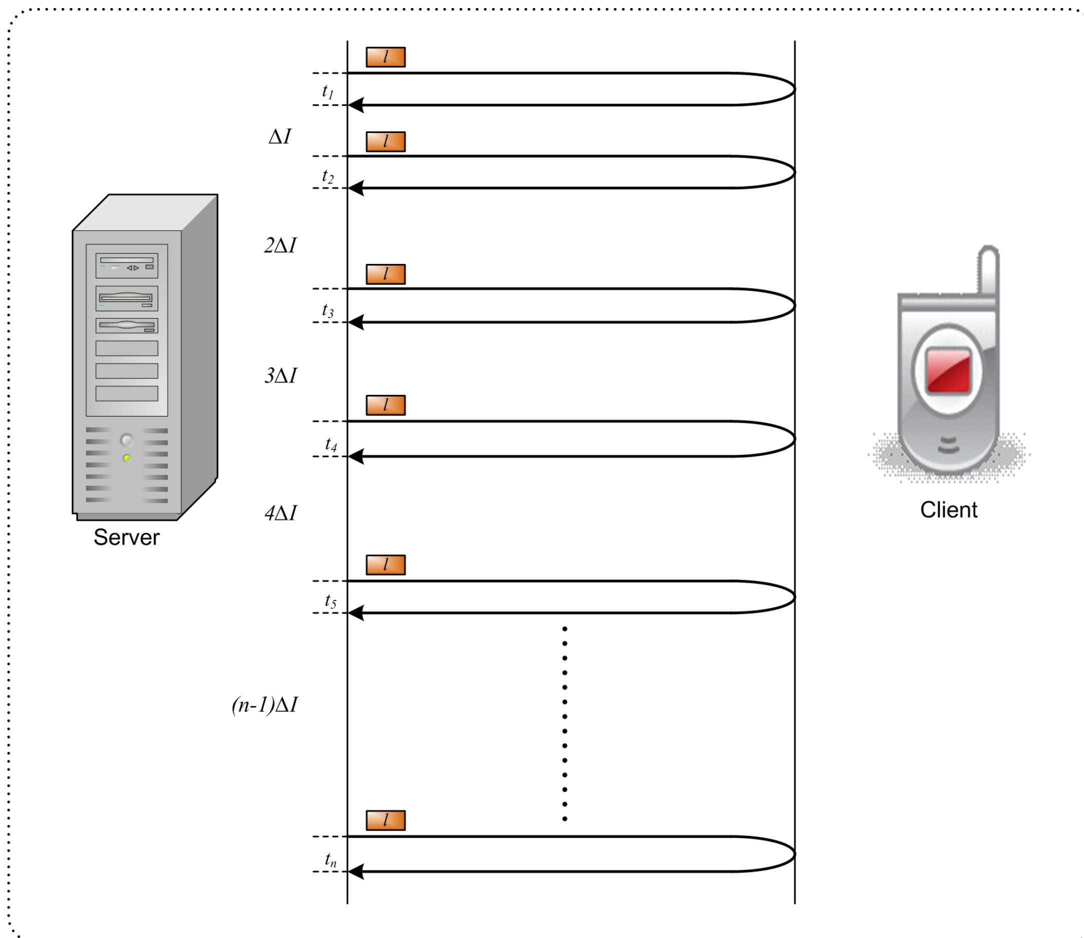

Figure 12.1: Response time test

enough latency to make a multiplayer mobile game satisfactory to play. However, every time the send interval is increased, the total time to send a packet is also increased with the same amount of time. The send interval with the lowest RTT may therefore not necessarily be the optimal send interval, if the delay is influencing the latency too much.

12.2 Transfer Speed

The transfer speed test measures the transfer time and transfer speed of the transmission of different sized packets from the server to a client and back. The test uses the same run and interval setup as the response time test. The initial size of the send packet and the size increase are defined in the test. The packet consists of an id, a separator character, an end-of-message character, and a number of 'x' characters to fill up the rest of the packet so that it has the desired size. The delay between each packet is defined according to the findings in the response time test, so that the packets are sent with an optimal interval. After the completion of each interval, the packet size is increased.

Figure 12.2 shows how the transfer speed test is performed. The packet to be sent is the orange

square and the size of the packet is the initial length, l_0, and the increment in size, Δl. The time the packet uses from the server and back is denoted as t_i where i represent the number of the packet, and the interval between sends is denoted ΔI.

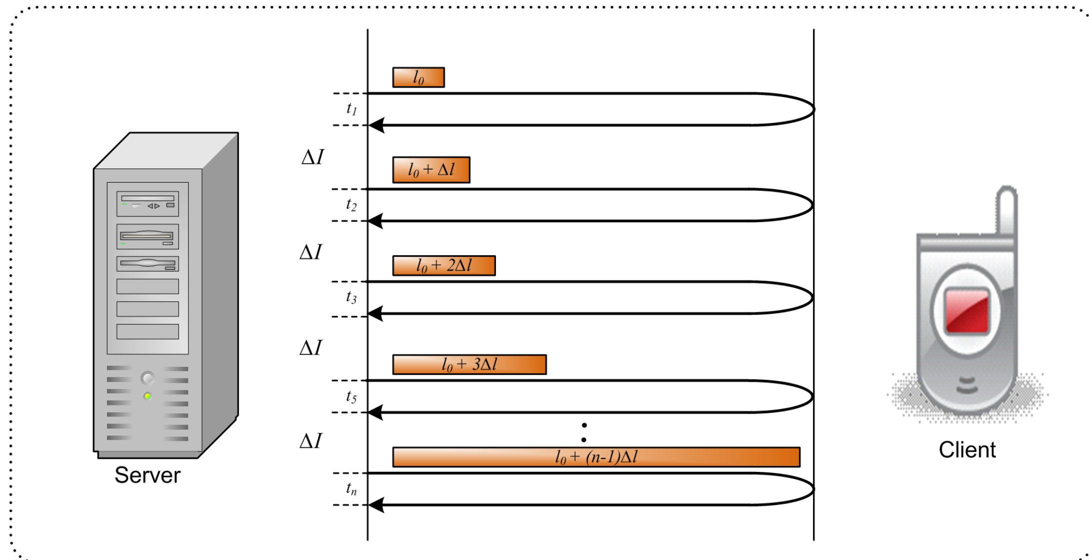

Figure 12.2: Transfer speed test

The purpose of the transfer speed is to compare the actual transfer speed of the different mobile network technology and the two transfer protocols. The increase in packet size is implemented to find a more correct transfer speed than by just sending the small packets used in the game. Each packet is sent from server to client and back, and the whole transmission is timed. This test also extract the highest and lowest times, and calculates the average for the rest. This average is used to calculate the average transfer speed by dividing the packet size with the transfer time.

Part IV

Test Results and Evaluation

Chapter 13

Test Results

In this chapter we present some statistical formulas that is used to calculate data from the test results as well as the test results themselves. In addition to the response time and transfer speed tests, a test of the data amount transferred in a normal game is performed. From this test the cost for playing the game for the user has been calculated. All the results and calculated data are summarized and concluded with at the end of the chapter. The test results used to generate the figures in this chapter can be found in Appendix E.

13.1 Formulas

From our test modules we have collected a large set of data. By themselves, these data do not provide much information, but statistical analysis of the values can give us the answers we seek. The most interesting statistical data from the test results are the average value, the variance, and the standard deviation. In this section, a short description of how to calculate these values is provided.

Average

With our data, the average formula is used to get an average response time for each interval. The formula is also used to find the average transfer time for each packet size in the transfer speed test. These average times can then be used as an indicator for how the expected average time would be for all future response and transfer time measurements. The formula for calculating the average time, $\bar{t}$, from a given set with n measured values, where the value of each measured time is t_i, is:

$$\bar{t} = \frac{1}{n}\sum_{i=1}^{n} t_i$$

For our calculation, we have used a slightly modified version of this formula. Since mobile networks can be a little unstable, we have removed the maximum and minimum value from the calculation of the average. This was done to find a average that is not influenced by values that are not representative for the real average. Hence, our formula for calculating the average times can be written as:

$$\bar{t} = \frac{1}{n-2}\left(\sum_{i=1}^{n} t_i - (t_{min} + t_{max})\right)$$

Variance

The variance is an indicator of how possible values are spread around the expected value, and shows the scale of these values [60]. The formula for calculating the variance, σ^2, with the same measurements as for the average value is:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(t_i - \bar{t})^2$$

Standard Deviation

The final value we well calculate from our test results, the standard deviation, is used to measure the spread of the values [60]. This value indicates how closely the measured values are located. If many values are far from the average value, the standard deviation is large, and if all the values are close to the average value, the standard deviation is small. In our case, this means that small variations in the measured times leads to a low standard deviation. Or, from another point of view, a small standard deviation indicates a stable and reliable network. The standard deviation is the square root of the variance, hence the formula for the standard deviation, σ, is simply:

$$\sigma = \sqrt{\sigma^2}$$

13.2 Response Time

The response time involved with each network technology depends on the time a minimum sized data packet takes from the server, to a client, and back to the client. For a real-time multiplayer game such as ours, this value has a very high significance, since most data packets sent are of a relatively small size. Thus, the response time values for each network technology indicates much about that particular network's suitability for a real-time multiplayer game. The test module we have used for testing the mobile networks' response time is further described in Section 12.1.

Figure 13.1 shows the results from the response time tests using different mobile network technologies (GPRS, EDGE, UMTS, and WLAN) and transport protocols (TCP and UDP), whereas Figure 13.2 shows the response times including the pause interval. This shows which interval is best suited for sending data with the different technologies. From the figures, one can see that UDP performs better than TCP on all networks. The figures also show that the send interval that provides the shortest response time is between 150 and 200 milliseconds, dependent of the mobile network technology used. WLAN has the lowest response time, followed by UMTS, EDGE, and GPRS. This coincide with the order of their performance specifications.

Figure 13.1 also shows that GPRS has far shorter response time with UDP compared to with TCP. This improvement is more significant with this mobile network technology than the rest. The difference between the two protocols on GPRS is around 1 second in average. This difference in response time will deliver two vastly different gameplays for the users. UDP also provides better response times with EDGE and GPRS than TCP does with UMTS at send

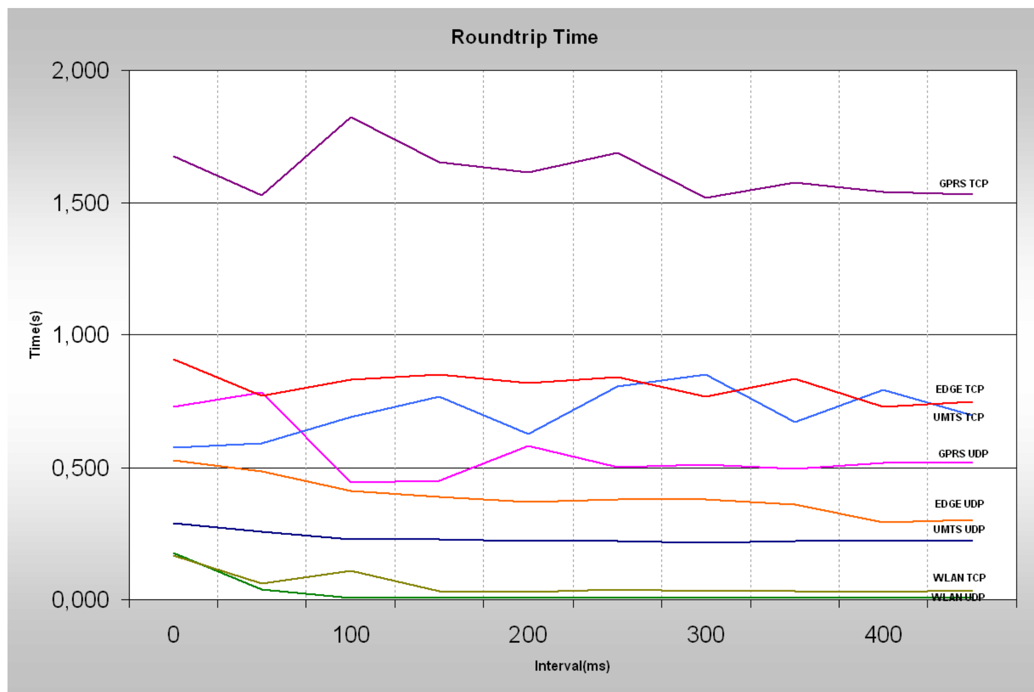

Figure 13.1: Measured response time

intervals larger than 75 ms. Of the three mobile network technologies, UMTS is the one with the highest expected performance. Hence, UDP clearly is the transport protocol that provides the best performance. TCP only delivers satisfactory response times with WLAN, because of WLAN's superior properties compared to the other network technologies.

The superiority of WLAN over the other network technology is evident in the two figures. The WLAN response time is below 0.2 seconds for all the send intervals, and with both transport protocols. The lowest response time for the rest of the networks is 0.217 seconds (UMTS with UDP). The UMTS response time is always less than that of EDGE using UDP. With TCP, the distance between the two is significantly less and the EDGE response time is even shorter than the UMTS response time in some intervals. GPRS with UDP is only in vicinity with EDGE between 100 ms and 150 ms. On the other send intervals, EDGE is closer to UMTS. With TCP, GPRS never has a response time below 1.5 seconds, which is too long a response time for a real-time multiplayer mobile game.

The average of the WLAN response times in all send intervals is around a tenth of the average of UMTS' response times with UDP. UMTS again has approximately 150 ms better average value than EDGE, which in turn has an average of 170 ms better than GPRS (with UDP). With TCP the order is the same, but the distances between the network technologies are different. The WLAN response time average is the lowest by far, almost a thirteenth of the UMTS response time. The difference between UMTS and EDGE has become smaller (100 ms in UMTS' advantage), whereas the GPRS response time average is around twice the EDGE response time average.

Figure 13.2 shows the response times including the send interval, i.e. the total response time from the previous packet is sent from the server to the server receives the return message from the client. This indicates the range of send intervals that will provide the shortest total

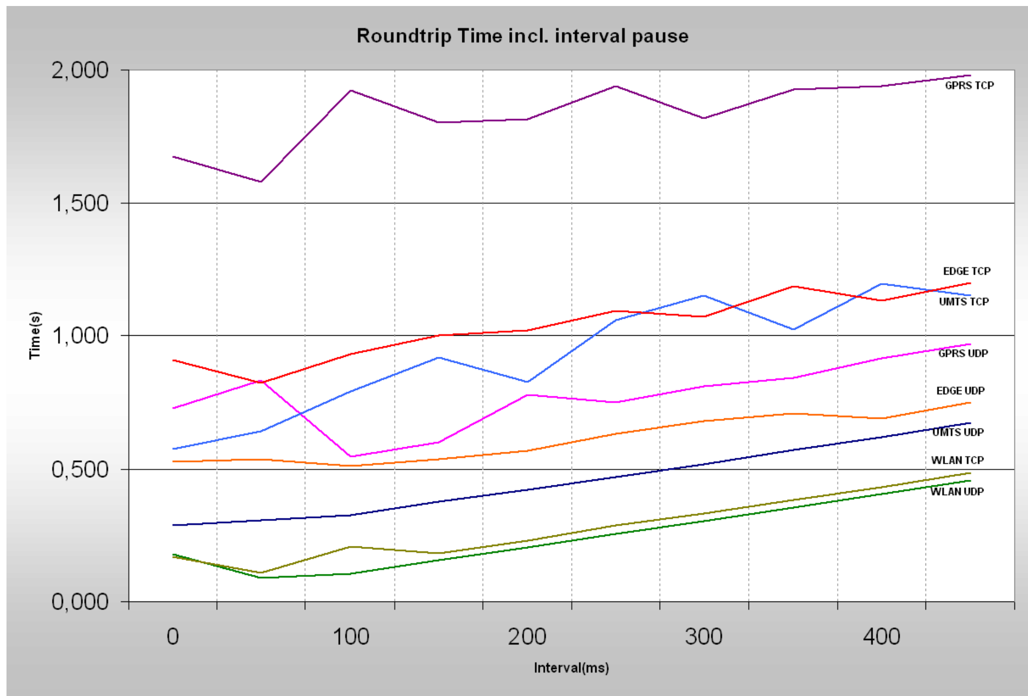

Figure 13.2: Measured response time including transmission interval

response time. The longer the pause interval is, the longer the total response time will be. From the figure, one can extract that pause interval values between 50 and 250 will provide the best total response time.

Statistical Data

The test results have been used to calculate the statistical values presented in Table 13.1. σ^2 represents the variance and σ represents the standard deviation. The data from the most relevant send intervals are used in the calculation. With send intervals at 100 ms and 250 ms, both an interval providing very good playability (100 ms), and an interval providing a lower cost involved with playing a game (250 ms) are considered. Shorter intervals than 100 ms will be very expensive, whereas higher intervals than 250 will not provide good enough playability.

Even though the calculated values are not unambiguous, in general they indicate that the 250 ms send interval has a lower variance(σ^2) than the 100 ms send interval on all mobile network technologies and both transport protocols. However, some values with TCP differ from this tendency. The only mobile network technology where TCP has lower statistical values than UDP is WLAN. The UDP variance values are lower than the TCP variance values on all networks, which also leads to the standard deviation(σ) values being lower. This means that the UDP test results have less variability than the TCP test results. UDP delivers response times that are more concentrated around the average, and therefore are more stable results with the same send interval.

Table 13.1: Statistical values of the response time test results with send interval 100 ms and 250 ms

Network/Protocol	σ^2 w/100 ms	σ w/100 ms	σ^2 w/250 ms	σ w/250 ms
WLAN/UDP	0.044 ms	6.66 ms	0.0014 ms	1.19 ms
UMTS/UDP	0.072 ms	8.49 ms	0.026 ms	5.12 ms
EDGE/UDP	5.609 ms	74.89 ms	0.761 ms	27.59 ms
GPRS/UDP	12.658 ms	112.51 ms	6.549 ms	80.92 ms
WLAN/TCP	0.011 ms	3.22 ms	0.012 ms	3.44 ms
UMTS/TCP	9.026 ms	95.01 ms	47.304 ms	217.49 ms
EDGE/TCP	16.041 ms	126.66 ms	14.692 ms	121.21 ms
GPRS/TCP	31.312 ms	176.96 ms	37.214 ms	192.91 ms

13.3 Transfer Speed

The transfer speed of a technology is a measurement of how much data the technology is able to transport per second. For a technology offering a high transfer speed, this means that larger data packets can be transmitted without loss of performance. This may have significant value for a real-time multiplayer game, since too low transfer speed may lead to some larger packets taking unacceptably long time before they reach their destination. The test module we have used for testing the different mobile network technologies' transfer speed is described in Section 12.2.

Figure 13.3 shows the results of the transfer speed tests, where short transfer time is best. In the presentation of the test results, we have chosen to use the measured transfer time for each packet size instead of the transfer speed. This is because the speed can easily be calculated from the transfer time, and the transfer time can be more directly related to the kind of games we are evaluating. From the figure, one can extract the time each mobile network uses to send a packet with a specific packet size from the server to the client and back. As seen in the figure, UDP provides more stable results than TCP. The mobile networks' order in terms of shortest transfer time are also as the expected order, based on their performance specifications.

When using UDP as the transport protocol, the size of the packet does not matter. In Figure 13.3, the transfer time is almost constant for all network technologies. WLAN has the lowest transfer time by far, whereas the UMTS transfer time is around 200-250 ms longer. The EDGE transfer time is another 100 ms longer, and the transfer time using GPRS is yet another 150 ms longer. The limit for a satisfactory transfer time depends on the send interval chosen and the amount of data to be sent to the client. With UDP, the four different mobile network technologies all have transfer times below or around 500 ms. This transfer time is measured from the server to the client and back, so the time from the server to the client can be expected to be half the measured transfer time. This means that the packet is received by the client within 250 ms. This is the same send interval that was considered a maximum satisfactory send interval in Section 13.2. Thus, by using UDP, the four network technologies all have the ability to deliver good enough transfer times with the most used packet sizes.

Using TCP, however, the packet size affects the transfer time more, except on WLAN where the transfer time is barely longer than with UDP. The transfer time with TCP vary more over the different packet sizes. With some packet sizes, EDGE is better than UMTS, but in average UMTS is better. GPRS with TCP has the most fluctuating transfer time. It also has the longest transfer time with all packet sizes. The average of all the packet sizes is more than

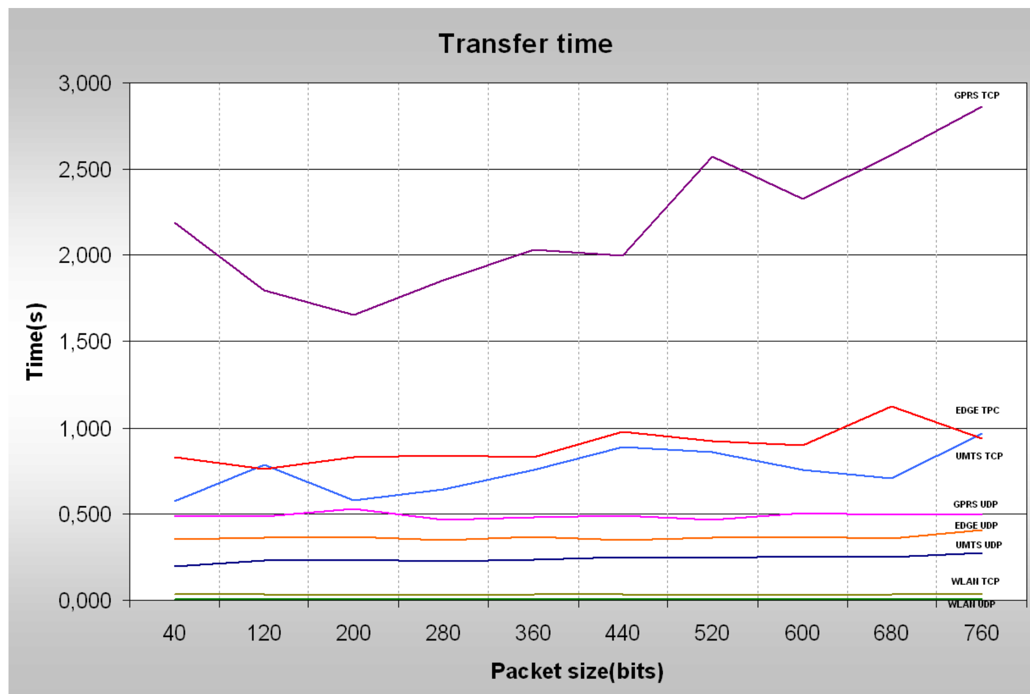

Figure 13.3: Measured transfer time

one second longer on GPRS with TCP than the second worst network technology's average transfer time with TCP(EDGE).

Statistical Data

The test results have been used to calculate the statistical values presented in Table 13.2. σ^2 represent the variance and σ represent the standard deviation. The data from the most relevant packet sizes are used in the calculation. With a packet size of 120 bits and 360 bits, both the most common packet size and the average packet size (related to the number of players) used in BrickBlock are considered.

Table 13.2: Statistical values of the transfer speed test results with packet size of 120 bits and 360 bits

Network/Protocol	σ^2 w/120 b	σ w/120 b	σ^2 w/360 b	σ w/360 b
WLAN/UDP	0.000 ms	0.483 ms	0.000 ms	0.422 ms
UMTS/UDP	0.161 ms	12.697 ms	0.159 ms	12.585 ms
EDGE/UDP	0.554 ms	23.533 ms	0.071 ms	8.407 ms
GPRS/UDP	4.974 ms	70.530 ms	9.417 ms	97.040 ms
WLAN/TCP	0.048 ms	6.957 ms	0.016 ms	4.062 ms
UMTS/TCP	107.803 ms	328.333 ms	22.753 ms	150.841 ms
EDGE/TCP	12.206 ms	110.481 ms	9.595 ms	97.956 ms
GPRS/TCP	53.251 ms	230.762 ms	39.722 ms	199.303 ms

Like the data from the response time test, the calculated values presented in Table 13.2 are not unambiguous. Still the variance(σ^2) seems to be lower with data size of 360 bits than with 120 bits on almost all network technologies. EDGE and GPRS have the most fluctuating statistical values. Their variance differs the most between the two data sizes compared to the other networks. From the measurements, one can extract that UDP has a lower variance and standard deviation(σ) than TCP.

13.4 Game Data Transfer

Users of multiplayer mobile games are charged by their service provider for the data they transmit and receive. The amount of data sent depends on the game type, whereas the amount of data received also depends on the number of players playing the game. The measurements of the data amounts are retrieved from the test mobile phone's own data transfer counter. The tests are performed using a Sony Ericsson W850i and Table 13.3 shows the properties of the phone setup. The player name and the screen resolution of the phone are sent to the server when a player connects, and is forwarded to the other clients when they connect to a common session. Therefore, the amount of data sent and received by a client depends on the player name's length and the phone's screen resolution. Also, the send interval pause, which determines how often the server sends data to the clients, affects the data amount. The shorter the send interval pause, the more data will be sent.

Table 13.3: The setup on the test phone

Property	**Value**
Player name	Player
Screen resolution	240x320
Mobile network	UMTS
Send interval	100 ms

In Section 5.3.4, a comparison of the mobile network technologies used in this project is presented. From the comparison in Table 5.3, the cost of downloading data with the different networks can be extracted. Downloading and uploading data with WLAN is free. This means that WLAN is not necessary to test in this cost comparison. WLAN is therefore disregarded. The other technologies, GPRS, EDGE, and UMTS, all have the same cost of 20 NOK/MB or approximately 0,02 NOK/kB. However, Telenor has a set maximum limit for data transfer costs for their subscriptions of 50 NOK per day [53]. Since the cost is the same, the mobile network chosen is irrelevant. UMTS is the default network on the test phone and is therefore chosen for this test.

The transport protocols are described in Section 5.4. TCP has a larger header than UDP and is connection-oriented. This means that communication is established before the data transfer to ensure that all packages are received. The communication establishment adds additional data to the send, which also applies to TCP's larger header (20 bytes in comparison with UDP's 8). Package losses happen more frequently with UDP. Clients may therefore receive fewer packages with the use of UDP instead of TCP in the game, which also theoretically may make UDP cheaper to use than TCP.

The game session tests have been performed by testing the data amounts sent and received when connecting to a server and session, when being idle in the lobby waiting for other players for 2 minutes (idle mode), when playing a normal game against 1 other player for 2 minutes,

and when playing against 3 other players for 2 minutes. An expected cost for a game against 7 opponents has been calculated based on these values. These tests provide information about the cost of playing the game with different numbers of users, as well as the average cost per minute of play. In the idle mode, the server sends out a request to the client to make sure that the client is still connected. The client must respond to the request. This generates some data transmission. Both the TCP (Table 13.4) and UDP (Table 13.5) protocols are tested since they have different packet headers, which means they send different amounts of data. The data amounts are measured in kilobytes (kB) and the related costs are measured in Norwegian kroner (NOK). The cost values are approximate values and are rounded to 3 decimals.

Table 13.4: Data amounts and cost with TCP

Type	Data Amounts	Cost	Cost/minute
Sent to create to session	465 bytes	0.009 NOK	
Received to create to session	563 bytes	0.010 NOK	
Total to create to session	1028 bytes	0.020 NOK	
Sent in idle mode	4118 bytes	0.079 NOK	0.039 NOK
Received in idle mode	7105 bytes	0.136 NOK	0.068 NOK
Total in idle mode	11223 bytes	0.214 NOK	0.107 NOK

Table 13.5: Data amounts and cost with UDP

Type	Data Amounts	Cost	Cost/minute
Sent to create to session	122 bytes	0.002 NOK	
Received to create to session	245 bytes	0.005 NOK	
Total to create to session	367 bytes	0.007 NOK	
Sent in idle mode	2486 bytes	0.047 NOK	0.024 NOK
Received in idle mode	2560 bytes	0.049 NOK	0.024 NOK
Total in idle mode	5046 bytes	0.096 NOK	0.048 NOK

Connecting to a server and creating a game session need a very small amount of data transfer to be completed. This applies to both transport protocols, but with UDP, approximately one third of the data is sent and received compared to TCP. When in idle mode, the data sent and received per minute is also very low. The cost of connecting to a session and then waiting for other players is therefore small. This is important since avoiding unnecessary costs in a game stage where the player does nothing may cause the game to attract players rather than the opposite. In idle mode, the cost using UDP is around half the cost with TCP. From the figures in Section 13.2 and Section 13.3, the difference in performance between TCP and UDP shows that the latter transport protocol is best suited for a real-time multiplayer mobile game. Also, by comparing Tables 13.4 and 13.5, one can see that with the use of TCP, more data is sent and received than with the use of UDP when connecting and creating a game session and in idle mode. Since UDP has so much better performance and is so much cheaper than TCP, we have chosen to only test the game sessions' data amounts with UDP (Table 13.6). The cost of connecting to the server and to a game session is included in the table. The creation of the game session is performed by one of the opponents. To keep the idle mode influence on the cost to a minimum, the game is started as soon as the test player is connected and ready.

As can be seen in Table 13.6, the clients receive more data than they send. This is because the server sends information about all the rest of the players to the clients. The sent data amounts vary because of the difference in force push commands sent. Pushing players around involves

Table 13.6: Data amounts and cost with UDP while playing the game

Type	Data Amounts	Cost	Cost/minute
Sent w/ 2 players	41794 bytes	0.797 NOK	0.399 NOK
Received w/ 2 players	63558 bytes	1.212 NOK	0.606 NOK
Total w/ 2 players	105352 bytes	2.009 NOK	1.005 NOK
Sent w/ 4 players	38650 bytes	0.738 NOK	0.369 NOK
Received w/ 4 players	85509 bytes	1.631 NOK	0.815 NOK
Total w/ 4 players	124159 bytes	2.369 NOK	1.184 NOK

sending more data to the server. However, players that do not move do not send position updates to the server, which in turn lowers the data amount sent from the server to the rest of the players. The more actions the player performs, the more data is sent to the server. The number of players is the main influence on the cost. Playing against one opponent costs just above 1 NOK per minute, whereas playing against 3 opponents costs slightly more. With more players in the game, force push commands are more likely to occur. Thus, increasing the cost. Also, with more players in the game, the position messages sent from the server to the client will be longer since all position updates are bundled into one message. This further increases the amount of received data.

With 8 players the client's sent and received data amounts will increase. The received data will increase the most, since the position messages will be increased by 28 bytes compared to the messages with 4 players. From the other test results and the logical hypothetic data amount increase, one can assume that the cost for each player when playing against 7 opponents will be between 1.50 and 2 NOK per minute.

The cost of playing the game in a real multiplayer setting may be considered too expensive. However, by using a longer send interval pause than the 100 ms used in this test, the cost can be decreased. Depending on the network technology, this will lead to reduced playability and game flow. While playing against 1 opponent, it will take almost exactly 50 minutes of play to reach Telenor's maximum data amount limit. Further playing time will be free of charge for the player. With 3 opponents this limit will be reached in less than 50 minutes of playing time.

13.5 Large Data Packets

We were a little surprised to find that the transfer time measured in Section 13.3 did not increase with increasing packet sizes. Motivated by this, we decided to run a new test using the same test module to determine if this was also true if the size of the data packets were further increased. The results from Section 13.2 and Section 13.3 clearly show that UDP over UMTS is best suited among the widely available network technologies. Because of this, the test with large data packets was only performed using UDP over UMTS. The graph in Figure 13.4 shows the measured results from this test.

As the figure shows, the transfer time is relatively stable between 200 ms and 300 ms independent of the packet size, up to a packet size of 11 240 bits. This further illustrates the trend indicated by the results discussed in Section 13.3. As long as UDP is used for transporting data, the transfer time is independent of the packet size. From this, we can conclude that in a multiplayer game, the size of the packets does not affect the time it takes for a packet sent

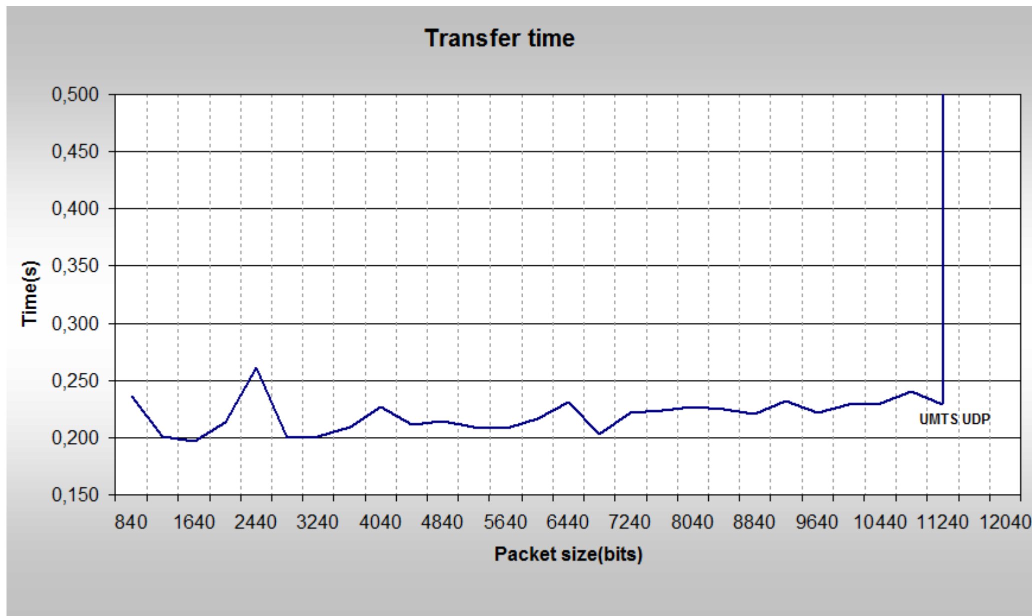

Figure 13.4: Measured transfer time with large data packets

from the server to reach the client (and vice versa). However, as discussed in Section 13.4, the size of the data packets directly affects the cost involved with playing such a game. Thus, even though the results from this test show that large data packets do not reduce the performance of a mobile multiplayer game, the cost of sending the data should still be an important factor of motivation for keeping the data packets as small and compact as possible.

At the right hand side of the graph in Figure 13.4, we see that when the size of the data packets exceeds 11 240 bits, the average transfer time of those packets increases infinitely. The reason for this is that data packets of this size are all lost. Even though not directly relevant for our concept game, we thought this to be an interesting find. This limit means that a game can never use data packets equal to or greater than this size, as it effectively results in all packets being lost without reaching their destinations. However, the test from the above figure used rather large increases in packet sizes. Thus, the exact limit for where all packets are lost does not clearly come forward from that test. To determine this limit, we ran a new test with size increases of 40 bits between 11 240 and 11 640 bits. This test showed us that the limit for the packet losses were between 11 560 and 11 640 bits. Finally, we ran a test using size increases of 8 bits (the least increment supported by our test module) with packet sizes between 11 560 and 11 640 bits. The results from this test are shown in the graph in Figure 13.5.

We expect that the small peak around packet sizes of 11 570 is a result of small variations in the network conditions when we ran the test. The interesting part in this figure is the graph's sudden break at 11 616 bits. In this test, this was the packet size above which all data packets were lost. Hence, the maximum data packet size that can be used in a multiplayer game for mobile phones using UDP as its transport protocol is 11 616 bits, or 1 452 bytes. For data packets containing more than these 1 452 bytes, the packet will have to be segmented into two or more smaller packets. Even though this is not necessary information for our prototype game, developers of other games or applications using UDP as transport protocol over UMTS should be aware of this limitation.

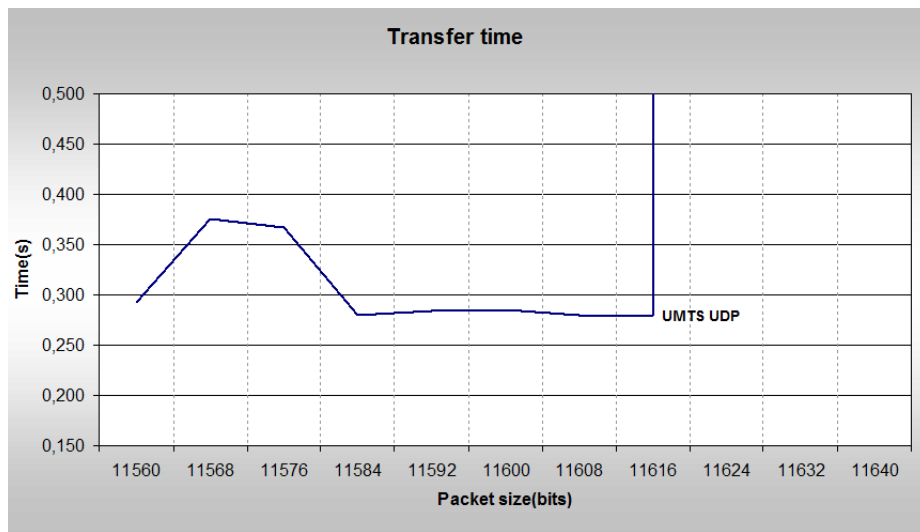

Figure 13.5: Locating the maximum packet size

13.6 Summary

As expected from the networks' specifications, WLAN has the shortest response time and fastest transfer speed, followed by UMTS, EDGE, and GPRS. Since WLAN is not widely available on mobile phones, whereas UMTS is, UMTS will be the most used mobile network by players playing the BrickBlock game. The interval between each send, i.e. the pause interval, has significant effect on the response time. The response time decreases with increases in the send pause. However, the send interval itself affects the total response time, so that it is not profitable to use too high pause intervals. Since more data will be transfered from the server to the clients with a short interval than with a long one, the cost of the data transmission increases with the decrease in send interval. Also, the cost increases with more players playing against each other. Low numbers of players and long send intervals will be cheapest, but this will also provide the least entertaining gameplay.

The shortest response times are found with a send interval between 100 and 250 ms. However, the differences between these intervals are large. With a send interval of 100 ms, the data updates would be be more frequent, and a game would be more real-time. Clients would receive position updates 10 times a second and the playability would be high. The downside of this send interval is its cost. With so frequent data updates, the cost of the game's data transfer would be high, since more data would have to be transfered. A send interval at 250 ms would be far cheaper, since the data updates would happen only four times per second. However, this would lead to less real-time feel in the game and reduced playability. A compromise is required to find the best performance with an affordable cost. The send interval could be chosen depending on the mobile network technology used. Because of the higher performance of UMTS compared to GPRS a send interval at 200 ms would be sufficient for UMTS, while too long for GPRS.

The transport protocols TCP and UDP deliver far different test results on the mobile networks. UDP yields better results with mobile network technologies with lower performance specifications than TCP does on technologies with higher specifications. The transfer time test indicates that the packet size has no influence on the transfer time when using UDP as

the transport protocol. This is further proved by the test sets where very large data packets are used. Because of this, all packets sent in a game would be transferred from the server to the client within a satisfactory time, independent of the size of the data packets, as long as UDP is used. This applies to all the mobile network technologies we have tested. TCP delivers more varying and longer transfer times. The only mobile network where TCP would deliver a suitable transfer time is WLAN, because of WLAN's superior properties compared to the other networks. Both the response time and the transfer time measurements suggest that UDP has the best performance.

In addition to providing better performance, UDP is cheaper to use than TCP because of a smaller header size. Another property with UDP that can influence the cost of data transfer is its unreliability. Packet losses happen more frequently with UDP than with TCP, which might decrease the cost. However, packet losses are best to avoid to ensure better data updates. When using UDP as transport protocol, the connecting to and creation of a game session cost less than 0.01 NOK. While the player is in idle mode, waiting for the game to start, the cost is less than 0.05 NOK per minute. With TCP, these costs are 0.02 NOK and 0.11 NOK per minute, respectively.

Playing the game on UMTS and UDP with a 100 ms send interval against one opponent will cost around 1 NOK per minute. The amount of received data increases with the number of players, because of the increased size of the position messages and more frequent force push commands. A possible solution to decrease the cost is to increase the send interval. This will decrease the number of packets send per second, and thus decrease the total amount of data received by the client. However, this will also decrease the real-time feel of the game. Without a clever business model or pricing model, a compromise between cost and playability has to be made.

Our final set of test results shows that there is a limit as to how large the UDP packets can be allowed to be. We measured this limit to be 1 452 bytes. If data packets larger than this are sent using UDP, all packets of that size will be lost before they reach their destination. As explained, even though this does not directly affect the game prototype developed in this project, it is an important limitation to keep in mind for other multiplayer mobile game that may need to send large data packets using UDP.

Chapter 14

Problems Encountered

During this project, we have run into several problems of varying degrees, as was to be expected. In this chapter, the most significant of the problems are described. For most of the problems, we have found a satisfactory solution. In these cases, the description of the problem and its solution can be used as a guideline and basis for solving problems on similar projects on later occasions. The problems that we were able to solve did not affect the result of our project beyond the time lost searching for the solution.

Unfortunately, there were also problems that we were not able to solve, or that we simply had to disregard. Some of these had little or no direct influence on our results, while others had a more noticeable impact. These problems are also described in this chapter. The problems that affected the result of our project, and their consequences will also be mentioned in succeeding chapters.

14.1 Java Related Problems

This section describes the problems we experienced that can be directly related to the Java programming language. As the section shows, these problems were related to the Java ME programming. Because of this, this section is most useful for mobile application developers.

14.1.1 The `Connection` Classes

As mentioned in Section 11.2.1, we use two different classes found in the Java ME API for TCP and UDP communication. These classes are `SocketConnection` and `UDPDatagramConnection`, respectively. According to the API, both these classes are implementations of the `Connection` interface [27]. Determining whether the communication object created when connecting to another entity is a `SocketConnection` or a `UDPDatagramConnection` should therefore be a simple task using Java's `instanceof` keyword. In our implementation we need to do this, as the waiting data have somewhat different formats dependent of which connection implementation currently in use.

When we tested our application on the emulators, this solution worked just as it was supposed to. However, we discovered a problem when we tested the application on our test phones. Apparently, Sony Ericsson's (and possibly other manufacturers') implementation of

the Communication classes differ from that specified in the Java ME API[1]. In Sony Ericsson's implementation, UDPDatagramConnection appears to be direct subclass of SocketConnection. The test (connection instanceof SocketConnection) will therefore always return true, independent of whether the connection in question is actually a SocketConnection or a UDPDatagramConnection object.

It is therefore important that the connection object is tested for being a UDPDatagramConnection object *before* it is tested for being a SocketConnection object. Listing 14.1 shows an example of this usage.

Listing 14.1: Use of instanceof on Connection objects

```
Connection connection = Connector.open( /* socket or datagram address */ );
if (connection instanceof UDPDatagramConnection) {
    // Do UDP-related procedures
}
else if (connection instanceof SocketConnection) {
    // Do TCP-related procedures
}
else {
    // The connection object is of unknown type. Handle this.
}
```

Since this solution solved the communication problem with all the mobile phones we have used for our testing, we have not looked further into the problem. However, if mobile phones from other manufacturers are used, or other communication protocols, similar issues may occur. As documentation for the different manufacturers' java implementation is very hard to come by, this experience can be used as a basis for trying to come up with similar solutions.

14.1.2 Using language level 5.0 with JWT 2.5

As mentioned, Java 5.0 and higher support generics and enums, whereas Java 1.4.2 does not. Since one of the requirements when installing JWT 2.5 is Java version 5.0 [26], we expected that JWT 2.5 also would support this. Such support would have been very useful in our implementation, as it would have meant that a lot of code could have been reused on both server and client without modifications. However, when we tried to use this language level for our client implementation we ran into several problems. These problems are described in this section, along with how and when they occurred.

StringBuilder not found
cannot access java.lang.StringBuilder

This was the error message that was displayed when we simply tried to set the language level of the compiler to 5.0, while keeping our code written in language level 1.4 (only functionality supported by Java version 1.4). The StringBuilder class is used to build strings using the '+' operator, for example by writing `String s = "Hello " + "World";`. When using the '+' operator on only String objects (as in the example), no error message was produced.

However, when another object or primitive other than a String is used, this error message was shown and the compilation failed. An example of this is the string `String s = "Number " + 1`. In a Java SE application, or a Java ME application compiled with language level 1.4, this kind of string building works just fine. But since the StringBuilder could not be accessed, it produced an error message in our case.

[1] We tried to find documentation from Sony Ericsson explaining this difference, but did not find any. Our description of the cause is therefore based on the explanation we find most likely.

As a workaround to this problem, we tried to use a `StringBuffer` object instead of the `StringBuilder`. Our technique was then as shown in Listing 14.2. This worked just fine, as long as we used this method in all places where we had previously used the '+' operator for building strings. The method `test()` would then write the string "*Number 1 is false*" if this was the only problem. Of course, a separate `StringBuffer` object can be created and used each time a string needs to be built, but creating this static method prevents having to write the same code many times.

Listing 14.2: Using a `StringBuffer` for building strings

```
1  public static String buildString(Object ... objects) {
2      StringBuffer buffer = new StringBuffer();
3      for (Object o : objects) {
4          buffer.append(o.toString());
5      }
6      return buffer.toString();
7  }
8
9  public void test() {
10     String s = buildString("Number ", 1, " is ", false);
11     System.out.println(s);
12 }
```

Compiler internal error.
Process terminated with exit code 4

Unfortunately, when using the above mentioned method, this error occurred. After trying several different ways to locate the reason for this error, we discovered that it occurs when sending primitives to the `buildString()` method without using object wrappers (also known as "boxing"). Hence, strings like `String s = buildString("Number ", 1)` produce this error. Instead, object wrappers need to be used for all primitives (int, boolean, byte, etc.). The test method from Listing 14.2 therefore has to be written as shown in Listing 14.3 to avoid this error.

Listing 14.3: Boxing primitives to avoid compiler error

```
9  public void test() {
10     String s = buildString("Number ", new Integer(1), " is ", Boolean.FALSE);
11     System.out.println(s);
12 }
```

Bad version information
ALERT: java/lang/ClassFormatError: Bad version information

When the above mentioned boxing was used to avoid passing primitives as objects, the compilation passed without problems, and we thought the problems were solved. However, when we tried to run the program on the emulator, this error message was printed, and the application closed. The only solution we could find to this was setting the language level of the compiler to 1.4, and we were back at the beginning.

Still, we were not able to find any *official* information on whether language level 5.0 is (or will be) supported by the JWT or not. So even though we had to give up and write our current client application in language level 1.4, changing this to 5.0 in the future should be a relatively simple task. Hopefully, this description of our problems can provide some help in this process.

14.1.3 Heap Address Error

We were able to avoid the error messages described in the previous section by simply using language level 1.4 for the client application instead of level 5.0. Unfortunately, there is another

error message that occurs from time to time when playing the game which we have not been able to work around: ***ALERT: Heap address is not four-byte aligned***. We are not sure why this error occurs, but we have noticed that the more players that are connected to a game, the more frequently the error message is displayed. Since we did not test the game with many connected players at the same time before late in the project, this error was discovered rather late.

Since the JVM on a mobile phone does not have a console like an emulator, we have detected this error while running at least one of the connected clients on emulator. When the error occurs, the emulator exits immediately. This does not happen on the mobile phones. However, we have also experienced that the mobile phones loses their connections to the server from time to time and are not able to reconnect. We suspect that this happens for the same reasons that the emulator shuts down.

Unfortunately, we have not been able to find any information on why this happens. We have tried searching on the Internet, but have not found any explanations. This problem therefore remains unresolved, and leads to the BrickBlock game being more unstable than we would have liked. Still, the error only happens occasionally, and only affects one player at the time. Thus, the remaining players may still continue their game session even if one of the players loses his connection because of this error.

14.2 Algorithmic Problems

While the previous section describes direct and concrete errors caused by programming errors and application problems, this section discusses the more abstract and vague algorithmic issues. A problem with such issues is that weaknesses in an algorithm does not necessarily lead to errors, but rather reduced quality of the application. Also, looking for a “perfect” algorithm may in many cases be futile, and in such cases, a limit has to be set for when the algorithm can be judged to be “good enough”. In this section, the algorithmic issues we ran into in our project is described. For those situations where we found a solution, this solution is presented. For problems we did not solve, a possible solution is proposed. Since these algorithms are related to the gameplay of BrickBlock, the section is most likely to be helpful for other game developers.

14.2.1 Movement Prediction

The algorithm we use for the movement prediction is described in Section 9.1. As explained, we use the simple movement prediction where each player simply keeps moving in the same direction until a new position update is received. Then, a new movement vector is calculated, and the player is moved along this movement vector. Most of the time, this movement prediction works satisfactory, and helps the game run smoothly on the players’ mobile phones. However, there are a couple of situations where the movement prediction algorithm does not work as well as we could have wished for.

Warping

As long as the player moves in straight lines most of the time, and does not constantly change direction, our movement prediction algorithm works very well. Unfortunately, the players do not necessarily move in straight lines all the time. If a player feels like it, he may change

direction as often as he likes. This may lead to much correction of that player's brick on the other clients' game boards. As shown in Table 9.1, in the worst case scenario, the warp distance can be as much as $2d$ (d is the distance moved). If this happens very often, the players will jump around on the game board each time new position updates are received, and trying to hit and push other players will be close to impossible.

This problem has two possible solutions. The simplest of these is minimizing the size of d. Since d is the distance the player moves between position updates, it can be reduced by simply sending position updates more often, or reducing the speed of the player. However, both of these methods have their downsides. If position updates are sent more often, the amount of data sent per game will increase correspondingly. As further explained in Section 13.4, this may not be desirable for the players, as it leads to a more expensive game to play.

On the other hand, reducing the player's speed leads to the players' bricks moving slower on the game board. This is very likely to decrease the fun involved with playing the game, as the game will be less hectic. For these reasons, it is important to find appropriate values for the frequency of position updates and the players' speed. Some warping will have to be allowed as a compromise.

The other solution to minimize the warping involved with movement prediction was discussed in Section 9.2, namely interpolation. This is a technique that reduces the amount of warping significantly, or removes it entirely. However, as the previous discussion concluded, interpolation and smooth turning is not suitable for the kind of game BrickBlock is desired to be. This solution can therefore be valuable for other games with a slightly different gameplay, but for our game prototype, we have decided not to make use of it.

The movement prediction is definitely a problem in our current implementation of BrickBlock, because of the warping caused by the prediction, but we have not found a completely satisfactory solution. For developers seeking to improve the BrickBlock game, looking into this problem and finding better solutions than those we have found could be worth the effort, since it would lead to a better and smoother gameflow than that currently in the game.

Detecting stopped players

Another problem related to our movement prediction occurs sometimes when a player stops moving. To avoid sending unnecessary position updates when the player is standing still, the client sends two equal positions when the player stops, and then waits for the player to start moving again before sending new position updates. The receiving clients then calculates the player's movement vector based on these positions. Since the positions are equal, the movement vector will be a 0-vector, and the player's brick will stand still with all clients. This method works very well most of the time.

However, since we use UDP for the communication, data packets are sometimes lost. If this happens with one (or both) of the equal position updates, the player's movement vector will never be detected to be 0, and the player will not stop moving. To make things worse, since position updates are not transmitted while the player is standing still, no new position updates are received by any of the other clients. Because of the movement prediction algorithm, the stopped player will therefore keep moving in the same direction on the other player's screen. Eventually, he will disappear through one of the walls. If one of the equal positions are lost on the way from the stopped player to the server, all connected clients will experience this. If the packet is lost on the way from the server to a client, only the receiver of the packets will be affected. Figure 14.1 shows how this problem looks on a client where a position packet has

been lost. The left image of each step shows a section of the display on the client who has missed the stop packet, whereas the right shows the player's actual position and movement.

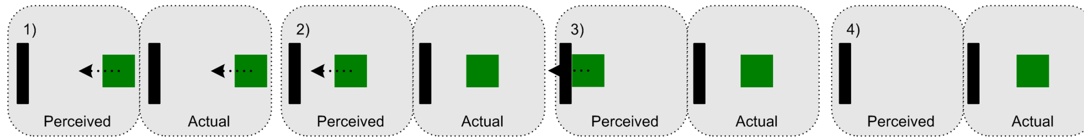

Figure 14.1: Illustration of the missed stop packet problem

The optimal solution to this problem would be implementing a mechanism for safe transmission of critical messages. This is implemented in TCP, but not in UDP. If this mechanism was available, the two position updates could be retransmitted until all connected clients had confirmed that the messages was received. As explained in Section 11.2, the `Communicator.sendMessage()` method contains a flag for requiring such confirmation, but the functionality is not implemented in this version of the framework.

Another, but much less elegant solution is never stopping the sending of position updates, or sending them less frequently. In this way, a few lost position updates is not that critical, since a position update will correct the player's position soon enough. Although this solves the problem in a satisfactory way, sending more information than necessary is not desirable. Since the users pay for every byte sent in most mobile networks, the amount of sent data should be minimized wherever possible.

A sort of middle way between the two-packet solution and always sending packets would be sending a larger number of position updates before stopping the transmission. In this way, the probability of at least two equal position updates reaching their destinations would seem higher. However, from our tests, we have discovered that one packet loss often is followed by several more packet losses. As a consequence of this, if two packets are lost, it is likely that five packets would also be lost. And again, the importance of reducing the cost where possible comes into play.

We have decided that for our prototype game, the problem with players not stopping because of lost packets is a non-critical problem. Firstly, such packet losses are rare, at least with the network conditions we have tested the game. Secondly, and more importantly, BrickBlock is not designed for static play. Players that are not moving can not push other players. At the same time, they are easy targets for other players seeking to push them into the trap. Combined, these two reasons show that this problem is not very likely to appear during a real game outside the testing environment. For other kinds of games though, and in poor network conditions, this may be a problem that needs careful consideration.

14.2.2 Pushing Other Players

The main goal of BrickBlock is pushing other players into the trap, and by doing this causing a negative point for the pushed player. Making this force pushing work in a satisfactory way has proved to be the biggest problem in our implementation. In the current implementation of BrickBlock, this is done by calculating the strength ratio between colliding players, and moving the weakest player in the strongest player's movement direction according to this ratio. However, because of the position updates that are continuously transmitted, this pushing does not work as well as could be desired. Each time a position update is sent from the pushed player, his position is corrected with all the other clients. Since these position updates are not

necessarily synchronized with the force vector, this correction may lead to pushed player being corrected to a position he has actually been pushed past. When this happens several times, the player will gradually be placed more and more under the pushing player. When a player has picked up a speed power up, this problem is even worse, as the difference between the positions in the updates are even greater. Figure 14.2 shows the problem with the force push calculation in a four-step illustration. In the figure, the blue player is the strongest player. Movement vectors are shown with dotted arrows, whereas the force vector is shown with a solid line.

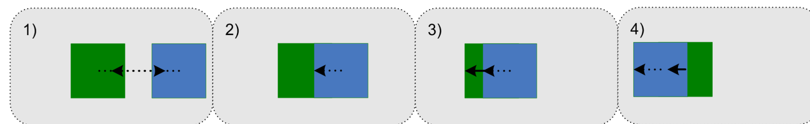

Figure 14.2: Illustration of the force push problem

The figure shows a section of the blue player's screen when he tries to push the green player. The steps illustrated in the figure are described in the following list.

1. The players are moving toward each other. Since no collision is detected, the players move forward.
2. Because of the network latency, the position update is received a little late from the green player, and he is placed partly under the blue player.
3. The strength ratio between the blue and the green player has been determined, and the blue player is the strongest. The green player is pushed in the direction of the solid line, which corresponds to the blue player's dotted movement vector. On the green player's client, the green player's position is calculated each time a position update is to be sent. When this position update is received by the blue player's client, the green player's position is corrected, and he is placed even more under the blue player.
4. When this goes on for a while, the blue player gradually moves completely over the green player. Eventually, the green player will be completely free from the blue player, on the opposite side of where he was initially pushed.

There are several possible methods that can be used to reduce this problem. However, we have not been able to find any solutions that solve the problem satisfactory. Two of these solutions are explained here.

Increase transmission frequency Once again, reducing the interval between each position update would reduce this problem. If position updates are sent more often, the deviance between the different clients' model of the game board would be reduced. Collisions would be detected closer to the same time with the involved players, and the weakest player would not be allowed to move toward the stronger. This increases the amount of data transmission in the game, and as a consequence increases the cost of the game. As previously discussed, this is only acceptable to a certain degree.

Forbid position updates from pushed players Another solution could be forbidding the pushed player to send position updates. This could be done by either the server or by the pushing player. The server would likely be the best alternative, as this would be the fastest way to notify all connected clients. However, the consequence of this would be that the pushed player could not move away from the weaker player. Still, a variant of this method, where only limited movement from the pushed player is allowed, would probably be the best way to improve the force push functionality of BrickBlock. For example, the server could calculate the positions of the pushed player based on his previous location,

his current movement vector, and the force vector received from the pushing player. Then the server could transmit this position to the other clients, instead of forwarding the position update from the pushed player immediately.

The current version of BrickBlock does not work as well as it should because of this problem. Players can push each other around the board, but not for long enough that it can be very well controlled. As mentioned at the beginning of this section, determining when an algorithm is satisfactory is more important than making it perfect. We find it unlikely that there is a way to make the pushing perfect, but that involving the server in calculating the pushed player's position could be worth checking out. This solution may take the force push algorithm one step closer to a satisfactory level. Still, we are unsure whether games requiring as extreme interaction between the game objects as BrickBlock really can be played in the mobile networks we have tested.

14.2.3 Difference in Player Speeds

When several players connect to a session with different mobile phones, they sometimes experience difference in the speed with which they move across the game board. Even though we have tried to find the cause for this issue, we have not been able to locate the problem with certainty. The problem is particularly visible if one of the players is connected with an emulator connected to the server via a LAN connection. Our first thought was therefore that the problem is related to the speed of the network the client uses. From this reasoning, a player connected to the server via a faster network connection would move faster across the gameboard than a player connected via a slower one.

However, the way we have implemented the movement in the framework, the player speed should be completely independent of the speed of the network communication. The player's movement is calculated in a local thread, and the thread responsible for sending player positions polls the current position at regular intervals. Hence, as long as the local thread runs at the same speed on the clients, the players will move the same amount of pixels each time the local thread calculates a new position. The movement of the player will always be proportional to the time interval between the polls as long as the player's speed is constant. This is illustrated in Figure 14.3. The client with the slower network connection will receive position updates from the server a little later than the client with the faster connection, but this will only have effect on the accuracy of the slower client's player models, not on the speed of the player objects.

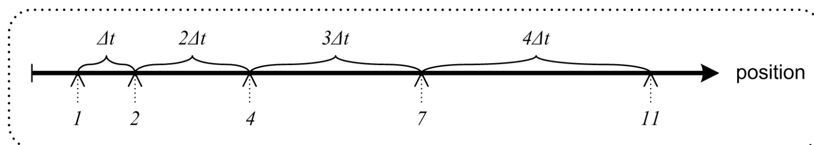

Figure 14.3: Player movement polling

This has led us to believe that the calculation speed of the clients is what results in these speed differences. With an FPS of 20 frames per second, the local player's brick is moved one pixel in the selected direction every 50 ms. If the client has to perform calculations that takes longer than these 50 ms, the time between each movement will also be increased. This results in a slower speed for the client with our current implementation. Since the processing power of a computer is far greater than that of a mobile phone, and mobile phones also may vary in terms of processing power, this seems to be a plausible reason.

The simple solution to this problem is reducing the FPS of the game enough that all clients are sure to complete their calculations before the player's brick is moved. If very slow clients are used to play the game, this solution may lead to the FPS having to be unacceptably low. Therefore, a better solution should be used. There are two possible solutions that are better suited for this problem. The first of these is assigning the responsibility of updating the player's position to a dedicated thread that runs independent of the other calculations. However, this solution may lead to collisions being detected a bit after the player has moved, which will reduce the responsivity of the game.

The other solution is using another unit of measure for the player's speed. Instead of using a value that indicates how far the player moves each time the position is updated, the speed can be measured in how far the player moves per time unit (for example per second). With this solution, the FPS will be automatically adjusted for slower clients, but the speed of all players will be equal independent of each client's available processing power. We have not had time to implement this solution in this version of the framework, but a possible implementation is given with pseudocode in Listing 14.4.

Listing 14.4: Using movement speed per time unit

```
1 speed = 20 pixels per second; // Corresponds to 20 FPS in the current implementation
2 while the game is running
3         calculated movement = speed / time passed since last movement;
4         store the current time;
5         perform calculations with calculated movement;
6         if calculated move is allowed
7                 move the calculated movement;
8         sleep for a given interval to avoid occupying too much resources;
```

14.3 Other Problems

During our implementation, we also discovered some strange issues that we do not have a good explanation for. These issues are related to our test results and our use of UDP as communication protocol, and are presented in this section.

14.3.1 Changed Test Results

When we first used our test modules to test the different networks with TCP and UDP as transport protocols, we got one set of results. Later in the project, we ran our tests again without changing the test modules. This time, the results of the tests were far more stable than those from the first tests. Also, the Round-Trip Times from the second set of test results were noticeably improved compared to the first set. This applied to all the tested networks (except WLAN), as well as both transport protocols. Our test results evaluated in Chapter 13 are the results derived from the second set of values, as we expect the latest test results to represent the current and future properties of the networks best.

We do not know why this significant change of network conditions suddenly occurred, and Telenor has not been able to explain this neither. Still, the values from both sets of test results show the same trends as to which networks are better and which are worse, and what transport protocol provides the fastest RTT. The values from both sets of test results are provided in Appendix E.

14.3.2 UDP Response

A curiosity we noticed while testing our game occurred when we tried to connect to a computer on the network without a running server. Since UDP is connectionless, we believed that data packets were only sent to the specified IP address, and then "forgotten" by the sender. However, when connecting to a computer without a running server, we experienced that our client application froze, seemingly waiting for a reply from the server. Even though the application still continued to run, it was obvious that another process was occupying a lot of resources in the background. When the server application was started on the specified computer, those resources seemed to be released, and the client application ran just as it is supposed to.

When trying to connect to a non-existing IP address, this did not happen. The application ran as supposed, with no background processes stealing resources. After the set time, the application displayed the notification that no connection could be established.

Since this is not a critical problem for the application, we have not looked further into it. When trying to connect to an IP address without a running server, nothing but a connection error notification is supposed to happen. Since the resources are released when the connection to the server is established, this curiosity has no impact on our applications. Still, it is a strange matter that could be worth checking into in order to gain a complete understanding of how the UDP communication works.

14.3.3 Randomly Generated UDP Packets

Another issue causing us a little consideration is that the server from time to another receives UDP packets that can not be interpreted like the regular UDP packets received from the client application. When trying to parse the contents of these packets using our parser, no contents can be extracted. However, we did not put very much effort into solving this problem. The server simply discards messages that it is not able to interpret. Once again, this is not an issue that affects our result noticeably, but it could be worth looking into to gain insight.

Chapter 15

Fulfillment of Requirements

In this chapter, the fulfillment of the requirements found in Chapter 10 is evaluated. For each of the requirements defined in the requirements chapter, a green tick means that the requirement is met, whereas a red cross means that it is not. Some of the requirements are only partly met. In these cases, the requirement is shown as not fulfilled, and the reason why it is not fully met is explained in the requirement's description.

15.1 Client Requirements

As mentioned, some of the requirements for the client application were also present in our depth study [29]. Therefore, some of the requirements were also already partly or completely fulfilled through the result of the depth study. Still, all the client's requirements specified in Chapter 10 are presented and evaluated in this section, to give a complete overview of which requirements have been fulfilled, and which have not.

15.1.1 Functional Requirements

A functional description of the client application's behavior was provided in Section 10.1.1, and in Table 10.1, this description was summarized in a list of numbered requirements. In this section, whether or not each of the client's functional requirements has been fulfilled is evaluated.

✔ **C-FR1:** *Each player may write his own player name, of up to 10 characters and numbers (including space).*
When the player starts the application, a startup screen is displayed where he can type his name.

✖ **C-FR2:** *Each player may select his own player color.*
This requirement is not completely met, but instead of letting each player select his own color, a random color is generated. The probability of each color supported by the phone is uniformly distributed, so it is not very likely that two players have the same color.

✔ **C-FR3:** *The server address and port can be selected when starting the application.*
The startup screen mentioned in C-FR1 also contain fields for entering server IP address and server port to connect to.

✔ **C-FR4:** *The communication protocol can be selected when starting the application.*
The same startup screen also contain radio buttons for selecting either TCP or UDP as communication protocol.

✔ **C-FR5:** *A player may start a new session at any time.*
When the player has connected to a server, a list of the active sessions is displayed. At the bottom of this list, the player can select to start a new session.

✔ **C-FR6:** *A player may join a session in progress if the session is not full.*
The session list mentioned in the previous point shows the active sessions' names, along with their number of connected players, and maximum number of players. As long as the number of connected players is lower than the maximum number of players, the player can connect to the session.

✔ **C-FR7:** *A player may refresh the session list.*
The session list screen contains a command for refreshing the session list. When this command is selected, a new version of the session list is downloaded and displayed.

✔ **C-FR8:** *A player may view the session's settings at any time.*
In the lobby screen, the player can select the command 'Settings'. When doing this, a list of the session's settings is displayed.

✔ **C-FR9:** *Any player in a session can change the settings for the session.*
In the settings screen mentioned in the previous point, the player can change the value of any setting. When selecting the 'OK' command, any changed settings are transmitted to the server and forwarded to all other connected players.

✔ **C-FR10:** *Any player in a session can start a new game if all the players are ready.*
Any player can select the 'Start' command from the lobby screen at any time. When this is done, the client application checks if all players have signaled that they are ready. If so, the server is told to start a game, and all connected players are notified.

✔ **C-FR11:** *All players connected to the session are listed in the lobby.*
Each player in the session is shown in the lobby screen, along with his ready status, name, color, and score.

✔ **C-FR12:** *If teams are enabled, the lobby sorts the players according to team.*
If teams are enabled, each team's players are grouped with their team members in the player list, with the team name above the team members.

✔ **C-FR13:** *A green or red dot signals if a player is ready for a new game or not.*
If a player has not signaled that he is ready, a red dot is shown before his name in the player list. Once he signals that he is ready, the color of the dot changes to green.

✔ **C-FR14:** *A player may enter the lobby at any time during a game without leaving the game.*
The game screen is displayed in full-screen mode, so the commands are not visible by default. However, the lobby view is available in some way, depending on the mobile phone being used. The 'Back' button leads back to the lobby view, and a 'Back' command is also available in the menu if this is displayed.

✔ **C-FR15:** *A list of the participating players can be viewed by pressing the FIRE button on the phone.*
Which button is assigned as the *FIRE* button depends on the mobile phone in use. However, for most mobile phones, the '5' dial has this functionality. Also, for mobile phones with a 5-way navigational key, pressing this navigational key also functions as pressing *FIRE*. When

this is done, a simplified version of the lobby view's player list is shown. This list contains the connected players' names and scores and is sorted descending by the scores.

✔ **C-FR16:** *The client must check for collisions each time the local player moves.*
Every time the player moves his brick, `GameView`'s abstract methods `checkObjectCollision()` and `checkPlayerCollision()` are called. If one of these methods detects a collision, the collision is handled accordingly. This also happens if the player is moved by external forces (i.e. he is pushed).

✔ **C-FR17:** *Collision with power up objects is handled by the client.*
In the BrickBlock implementation of the framework, object collisions are handled in the implementation of the abstract `handleObjectCollision()` method in the `BBBoard` class. If the object causing the collision is a power up object, the server is notified that such a collision has occurred. If so, the server notifies all connected clients that the player has picked up the power up, if it was available. This corresponds to the procedure discussed in Chapter 8.

✔ **C-FR18:** *Collision with trap is detected by the client.*
The procedure for handling trap collisions equals that mentioned in the previous point. If a trap collision is detected, the server is notified, and actions are taken according to the discussion in Chapter 8.

✔ **C-FR19:** *A player dies if his brick touches the trap.*
This requirement is fulfilled through the server's handling of trap collisions. If the server is notified that a player has collided with the trap, the score of the relevant player is reduced with 1, and the player is moved to one of the game board's corners.

✔ **C-FR20:** *A player can only be pushed by an equally strong or stronger player.*
When a player collision is detected by the `checkCollision()` method, this requirement is met through the implementation of the abstract `handlePlayerCollision()` method in `BBBoard`. Here, the strength ratio between the players is calculated, and a movement toward the other player is only allowed if the ratio is equal to or larger than 1.

✔ **C-FR21:** *The game board is updated every 20 ms.*
In the `LocalThread` inner class in `AbstractGame`, the local player is moved every 20 ms if he has pressed a movement key, thus fulfilling this requirement.

✔ **C-FR22:** *The position is polled and sent every 100 ms.*
This requirement is met in the `SenderThread` inner class in `AbstractGame`. Every 100 ms, the player's position is sent to the server, if he has moved since the last time his position was sent.

✔ **C-FR23:** *A prediction algorithm is used to approximate each player's position between updates.*
Every time the local player receives a position update from a remote player, the player's movement vector is calculated in the `AbstractPlayer.calculateMovementVector()` method. This movement vector is then used to move the player the appropriate distance every time the game board is updated.

✔ **C-FR24:** *Player positions are only sent while the player is moving.*
As described in CFR-24, the player's position is sent to the server by the `PositionThread` in `AbstractGame`. Before this position is sent, whether the player has moved or not since the last position transmission is calculated. If the position has not changed, the position is not sent. (Equal positions are sent exactly twice, as explained in Section 14.2, to let the other clients detect that the player has actually stopped).

✔ **C-FR25:** *When the game is over, the player is taken back to the lobby.*
The server is responsible for detecting when a game is over, and notifying all clients connected

to the session. When such a notification is received, an alert is displayed for three seconds, showing the reason why the game was stopped. All object lists in the `BBBoard` class are then cleared, and the client displays the lobby screen, with the connected players' final scores for the game.

15.1.2 Non-functional Requirements

The client's non-functional requirements were derived and presented in Section 10.1.2. In this section, each of these requirements are listed, along with an evaluation of whether or not the requirement has been fulfilled.

For the requirements applying to the usability of the application (C-NR1, C-NR2, and C-NR3), we have used a test group consisting of five persons. These persons have tried playing the BrickBlock game without any instructions or explanation beyond the rules of the game. For the other non-functional requirements, we have evaluated the fulfillment of each requirement based on test results and the architecture of the client application.

✔ **C-NR1:** *The client shall be understandable and easy to use within 2 minutes of play without explanation beyond that provided in the game rules.*
Our test candidates have tried playing the game, and have understood how to play the game immediately without further explanation.

✔ **C-NR2:** *The user interface shall be easy to comprehend and interact with.*
The same test candidates mentioned in the previous point also understood the functionality of the different commands and buttons immediately when testing the application.

✔ **C-NR3:** *The client shall give understandable feedback to the user's actions.*
Whenever the user performs an action, the client responds accordingly immediately. Examples of such actions are changing the values of settings or the player's ready status in the client's pre-game state. In the in-game state, a user action results in the user's brick moving according to the provided input.

✔ **C-NR4:** *The client shall send the player action to the server immediately.*
All user actions except movement are sent to the server immediately after the action has taken place. Since user movement is expected to happen continuously throughout a game, the user position is sent with regular intervals instead of immediately to reduce the user cost involved with playing the game. As long as these intervals are not too long (100 ms in the current implementation), the position updates are also sent quickly enough that they will seem immediate to the players playing the game.

✔ **C-NR5:** *The data sent from the client should be limited in size to ensure the information is received fast on the server, i.e. within 0.5 seconds.*
In a game with 20 active players, the longest possible message to send will be a message containing a position update for all players. Such a message will have a size of (4 B + 20 × 7 B = 144 B =) 1152 b. From our test results with large packet sizes found in Chapter 13.5, we see that the response time is not noticeably reduced with data packets of this size. Hence, data sent from the client will always be small enough that the server receives the data within 0.5 seconds after transmission from the client, as long as UDP is used for the transport.

✔ **C-NR6:** *The client shall handle faults in way that keeps the user unaware of the fault, as well as letting the game continue running.*
Such faults can be caused by illegal messages received from the server or lost packets. Through try-catch statements and evaluation of the state of the client's objects, such faults are caught

and handled before they can affect the game. When such a fault occurs, the client ignores the fault and keeps on running as if the fault never occurred.

✔ **C-NR7:** *A developer shall be able to add content to the client or change existing content without side effects on the rest of the application.*
Support for teams was added after the other functionality was implemented in the application. This was done by only adding code to the `AbstractPlayer`, `AbstractModel`, and `Utils` classes, and did not in any way affect the already existing functionality of the application.

Since the server handles settings and most of the game control, very little modification should be necessary for the client in order to add functionality to the application. The only part that may need modification on the client side are the graphical components, and these are easily extended without risk of compromising other existing functionality.

✔ **C-NR8:** *The client shall be usable as a basis for more advanced gaming concepts.*
Since the client is separated into a framework and a BrickBlock part, extending the framework is very simple. The basic functionality like the lobby screen and basic movement is already implemented in the framework. Beyond this, developers are free to include more advanced graphics and new game rules by extending the abstract methods found in the framework.

✔ **C-NR9:** *The client must support the transport protocols TCP and UDP on different network technologies.*
Both TCP and UDP are supported by the client by default, and which transport protocol to use can be selected when starting the client application. These protocols are independent of the network technology being used, and both TCP and UDP can be used for all mobile network technologies.

✘ **C-NR10:** *Variables for adjusting the operational speeds of the client, such as FPS and send interval, shall be stored in a common class.*
We have not gathered these variables in a common class as the requirement states, because other, more urgent tasks have been prioritized. Still, all such variables are declared static in the top of the classes where they are used.

15.2 Server Requirements

Unlike the client's requirements, the server's requirements had to be completely defined in this project, as the previous version of the framework and BrickBlock used a P2P architecture. This section contains a description of whether or not the server's requirements have been met in our server implementation.

15.2.1 Functional Requirements

Like the client's functional requirements, the server's functional requirements were derived through a functional description of the server's behavior in Section 10.2.1. This functionality was summarized into formal requirements in Table 10.2. In this section, the fulfillment of each of the server's functional requirements is evaluated corresponding to the evaluation of the client's functional requirements.

✔ **S-FR1:** *The server is implemented in Java SE.*
The server application is written using JDK 6.0.

✔ **S-FR2:** *The server can be run on any computer with an Internet connection.*
Since the server, as stated in the previous point, is implemented using JDK 6.0, the server will be able to run on any computer that has Java Runtime Environment (JRE) 6.0 installed, and is connected to the Internet.

✔ **S-FR3:** *The server can run multiple simultaneous sessions.*
The server contains a list of the currently running sessions. This list can easily be extended with more sessions, and each session contains its own game threads. Hence, the server can run many simultaneous sessions.

✔ **S-FR4:** *The server sends alive requests with regular intervals.*
The server sends alive requests to all clients with an interval of two seconds between each request. The responsibility for this is delegated to the inner class `ConnectionChecker` in `AbstractSession`.

✔ **S-FR5:** *A client failing to respond to alive requests is removed from the session.*
If a client fails to respond to ten succeeding requests, he is removed from the session and the server. All other connected clients are then notified of the disconnection.

✔ **S-FR6:** *Changed settings are immediately forwarded to all clients.*
When a changed setting is received from a client, the setting is immediately forwarded.

✔ **S-FR7:** *Start game commands are immediately forwarded to all clients.*
When a start game command is received from a client, the start game command is immediately forwarded to each client, along with the size of the game board and the player's start position.

✔ **S-FR8:** *The server generates a trap position when a new game starts.*
The trap position is generated by the `BBEventHandler` thread, which is the BrickBlock server's implementation of the `EventHandler` interface. When this thread is started, the trap position is generated and transmitted to all connected clients.

✔ **S-FR9:** *The server generates player positions when a new game starts.*
The player positions are also generated by the `BBEventHandler`, which makes sure that no two players occupy the same position. As described in point S-FR7, the initial player positions are transmitted along with the start game command.

✔ **S-FR10:** *The server handles generation of power up objects.*
The `BBEventHandler` is also responsible for generating power up objects. The `BBEventHandler` generates new power up objects continuously through a game, and each of the three kinds of power up objects have equal probability of being generated next.

✔ **S-FR11:** *The server generates power up objects with irregular intervals.*
A new power up object is generated by the `BBEventHandler` with an interval of between 5 and 15 seconds between each generation.

✔ **S-FR12:** *The server notifies all participants when a player has picked up a power up object.*
When the server receives a notification that a player has collided with a power up object, the `BBEventHandler` first checks if this power up is still available in its `fireEventOccurred()` method. If the power up is still available, the server notifies all players (including the player that picked up the power up) that the specified power up object has been picked up by the specified player.

✘ **S-FR13:** *When a player dies, he is placed (close to) one of the board's corners.*
When a player dies, he will be placed in one of the four corners as long as one is unoccupied. However, instead of placing the player close to one of the corners if the corners are all taken, the player is simply placed in a random unoccupied position on the game board. This was done

to reuse the generate random position method, and even though it is not an optimal solution, we think it is sufficient.

✔ **S-FR14:** *Updated player positions are transmitted in batches with regular intervals.*
The server's `GameThread` is responsible for storing the connected player's positions and transmitting these with regular intervals. Every time a position update is received from a client, the player's position in the position list is updated. Then, every 100 ms, the list of positions is used to create a position message containing the position of all players that have moved since the last time such a message was created. This position bundle is then sent to all connected players.

✔ **S-FR15:** *The server keeps track of the settings, and notifies all participants when the game is over.*
The server's `EventHandler` implementation also compares the game state to the settings in its `checkSettings()` method. If a limit specified through the settings is reached, the server sends a notification to all connected clients that the current game is over. This notification also contains the reason for why the game was stopped.

✔ **S-FR16:** *If all players disconnect from the session, the server closes the session.*
When the last player connected to a session disconnects from the session, there is no reason why the session should still be running and occupying resources. Every time a player disconnects from a session, the number of remaining players is therefore evaluated in the `AbstractSession.handleDisconnect()` method. If this number equals zero, the session's resources are released, and the session is closed.

✔ **S-FR17:** *When a game is over, the server releases resources and stops threads related to that game.*
A game is ended when the game's state reaches a limit defined in the session's settings, as described in S-FR15. When this happens, the session stops its two running threads; the `GameThread` and the `EventHandler`, and releases the resources occupied by these threads. This is done in `AbstractSession`'s `stopGame()` method.

15.2.2 Non-functional Requirements

In this section, the fulfillment of the non-functional requirements of the server are discussed and evaluated. These non-functional requirements were first presented in Section 10.2.2. Like the previous sections, the fulfillment of each requirement is represented by a green tick representing a fulfilled requirement, and a red cross representing a non-fulfilled or only partly fulfilled requirement. The fulfillment of non-functional requirements of the server was evaluated by simulating game events, inspecting and analyzing the code, and running performance tests. The experience gained from our test sessions mentioned in Section 15.1.1 was also taken into account in these evaluations.

✖ **S-NR1:** *The players may connect to the game in progress at any time.*
A player may connect to a session at any time. However, connecting to a running game is not fully supported. Instead of being able to join a game in progress, the player must wait in the lobby until the running game is completed. He will then be able to participate in the next game that is started.

✔ **S-NR2:** *The players may disconnect from the game in progress at any time.*
Whenever a player disconnects from a session, a disconnect command is sent to the server. If this disconnection notification is successfully received by the server, the server removes the player from its player lists, and notifies all other clients connected to the session about the

disconnection. For the player disconnecting from the session, the disconnection will appear to happen immediately, and the application will be closed.

✔ **S-NR3:** *The server shall handle unexpected disconnections without affecting the game.*
An unexpected disconnection may occur if the mobile phone shuts down before it is able to send its disconnection notification or if the notification is lost on its way to the server. In these cases, the server will notice that the client has been disconnected through its alive requests. When the server notices that a client fails to respond to several succeeding alive requests, it simply removes the client from the session and notifies all the other connected clients. The game then keeps on running as if the client was never connected to the session.

✔ **S-NR4:** *The server shall handle faults in a way that keeps the user unaware of the fault, as well as letting the game continue running.*
Like on the client, such faults can be illegal messages or lost packets. Through try-catch statements and evaluation of the objects used by the server, such faults are caught and handled before they can affect the game. When such a fault occurs, the server simply ignores the fault and keeps on running as if the fault never occurred.

✘ **S-NR5:** *The server shall be operational 99% of the time, i.e. low mean time to repair.*
If a critical error occurs despite of the fault handling described in the previous point, the server has to be manually restarted. Even though such a restart by itself is very fast, the time elapsed before the restart is carried out depends on the person(s) operating the server. Hence, if the server is not continuously observed, the mean time to repair may be quite long if an error occurs. To improve this, an automated mechanism for detecting critical errors and restarting the server will have to be implemented.

Even though the server in principle should be stable enough to avoid such critical errors, we have not had the possibility to test the stability of the server for a long period with many connected clients. Because of this, we can not say with certainty that critical errors never occur. Thus, we can not guarantee that this requirement is fulfilled.

✔ **S-NR6:** *The data sent from the server should be limited in size, while containing as much information as possible, to ensure fast data updates on the clients and low cost for the player.*
The messages sent from the server consist of a three-letter action identifier, an optional player id field, a list of values, and a end-of-message character, as described in Section 11.2. Where possible, several values are bundled into one message, as for example the position updates and the start game commands. In this way, we ensure that no redundant information is sent, and that the messages are (close to) as short as they possibly can be, while still being logical in their structure.

✔ **S-NR7:** *The server's in-game generating and calculating tasks shall be completed correctly within 10 ms to ensure a fast and responsive server.*
The most critical in-game calculations on the server are performed by the session's `GameThread` and `BBEventHandler` threads. Of these calculations, the most time-consuming calculation is the generation of the bundled position message. We have tested the calculation time for this task using 50 dummy players with random positions, running the calculation 1 000 000 times. (The implementation of this test can be found in the `main()` method in `GameThread`.) The result of the test was that the maximum elapsed time of such a generation was *6.2 ms* and the average elapsed time was *0.023 ms*. These results show that this requirement is met under these circumstances. In a real game, the number of players is likely to be less than 50, and the calculation time will therefore be even shorter.

✔ **S-NR8:** *A developer may add or change functionality without causing other functionality to stop working.*

The functionality for team support was added after the rest of the functionality of the framework was implemented. This was done by adding the “Teams enabled” setting. When implementing this functionality, the classes having to be modified were the `Setting`, `SettingsList`, and `PlayerPanel` classes. Adding this functionality did not affect the other functionality of the server in any way.

Likewise, adding or changing the in-game functionality can be done by modifying the contents of the `BBEventHandler` class, and will not affect the already existing functionality of the server.

✔ **S-NR9:** *The server must support the transport protocols TCP and UDP on different network technologies.*
The server contains support for both TCP and UDP, and which one of these to use on the server is specified as an argument when starting the server (see Appendix B). The network technology in use for the connected clients is completely independent of the choice of transport protocol, and as long as the mobile phone is connected to a mobile network, any of these two protocols can be used.

✖ **S-NR10:** *Values determining the operational speeds of the server, such as object generation and send intervals, shall be declared in an XML configuration file.*
Like the corresponding client non-functional requirement, CNR-10, this requirement has been disregarded because of more important tasks. Still, as on the client side, static variables for adjusting these values are declared at the top of all classes where they are used.

Chapter 16

Method Evaluation

In this project, we have used several different methods for answering our research questions and for developing our prototype game and framework. These methods were defined and described in Chapter 2 and Chapter 3. This chapter provides an evaluation of these methods, and the effect our choice of methods has had on the project's results.

16.1 Research Methods

As explained in Section 2.2, we have used several research approaches and methods to answer the project's research questions in the best possible way. Since some of the research questions were quite different in nature, different approaches and methods have been necessary. The following list contains an evaluation of each of those research approaches and methods, and whether or not they have provided the needed basis for answering our research questions satisfactory. At the end of this section, our selection of research approaches and methods as a whole is evaluated.

1. **The Engineering Method**
   To obtain data for three of the research questions, we made use of the *Engineering* method. This was done by implementing our prototype game in incremental tasks, and evaluating the behavior of the game after each iteration. We also recorded problems that we ran into during these iterations, and considered why these problems occurred and how they could be solved. Through this process, we obtained a basis that could be used to provide some answers to our research questions. For answering research questions of this kind, the *Engineering* method has proved to be very useful.

2. **The Empirical Method**
   The second research question was of a somewhat different nature than the other three, in that it required very specific and objective data measurements to answer in a satisfactory way. To obtain this data, we decided to make use of the *Empirical* method. This was done by implementing two test modules that were used to measure different aspects with the different mobile network technologies. The data obtained from these test modules could then be evaluated and used to answer that specific research question in an objective and unbiased way. For a research question of this nature, the Empirical Method proved to be a very useful method, and it provided a better basis for answering the question than the Engineering Method would have done.

3. **Lessons Learned**
   Since this master thesis is based on the results and experience from our previous depth study [29], this method would indirectly have been used even if we had not specified it among our research methods. In this project, we have used that experience as a background and guideline for our development of a prototype game. The problems previously encountered could be predicted and avoided based on this experience. Furthermore, from the lessons learned in the previous project, we had a deeper understanding of the problems at hand and could, to some degree, predict part of the answers to some of the research questions. This particularly applied to research questions 1, 3, and 4.

4. **Literature Search**
   In Chapter 6, three online multiplayer mobile games related to our prototype game are presented. Even though the documentation for these games was sparse, we were able to find some information that could be related to the development of our game. Also, we have been able to extract useful information from the other resources we have used throughout this project. This has provided us with a platform on which to base the work of this project.

5. **Simulation**
   The final research method mentioned in Section 2.2 is the *Simulation* method. Since mobile networks are available across the entire country, and there is an enormous amount of potential users for such a service, we had to limit the size of our simulation model to a more manageable size. We have then used the results from using this model as an indicator of the situation in the real environment. This method has been particularly useful to answer research question 2. However, question 4 has also been partly answered by using a small test group to simulate the potential users of such a game.

All five of our selected research methods have been very useful in order to obtain answers to our research questions. By themselves, each of the research methods has provided data that we could use to partly answer questions. Combined, we feel that the data collected from the research methods are sufficient to answer all of our research questions in a satisfactory way. Answers to each of the research questions are provided in Chapter 18.

16.2 Development Methods

In Section 3.1, we described the development method we would use for this project as a combination of the eXtreme Programming and the Unified Process development methods. Extracting key practices from different development methods in this way may allows us to utilize the best parts of each method. However, since our new development method is not fully neither one nor the other of the original methods, this approach also carry some risk. These methods have been developed through experience, and removing one or more practices, even if they seem unnecessary, may affect the efficiency of the other practices. In this section, we will evaluate each of the practices we decided to use in Chapter 3.1, and decide whether the practice has proved to be useful or not. Also, we will evaluate the development method as a whole, and determine the usefulness of that specific method for a project such as this.

In the following list, the ten practices selected for our development are presented corresponding to the list found in Section 3.1. How we have used that particular practice, and the effect the practice has had in the project is described.

1. **Simple design**
   As explained, the goal for this project was not creating a complex and advanced applica-

tion, but rather a simple implementation to test to what degree today's mobile network technologies are suitable for a real-time multiplayer mobile game. Sticking to a simple design has helped us avoid the risk of implementing too much "nice-to-have" functionality. Instead, only the necessary functionality have been implemented to create an application simple enough so evaluation of the aspects relevant for this project have been possible. The main relevant aspects have been network performance in a real-time multiplayer mobile game. In other words, this practice has been very useful for in the project's development process.

2. **Pair programming**
The use of the pair programming practice in this project is a slight modification of that defined in the XP method [46], in that we have not used it for the entire development process, but rather the harder, non-trivial issues. For this kind of issues, this practice has proved to be a very useful tool, by enabling us to utilize the knowledge and experience of both team members both to find solutions to the problems and to detect flaws in the code immediately. Furthermore, the choice to disregard the practice for the more trivial tasks has allowed us to complete these kinds of tasks quickly and efficiently. Even though this may have led to some small slips in the code, those errors have been quickly detected and corrected, and we believe this has been more effective than what would be the case if pair programming had been used for the entire implementation. All in all, our use of the pair programming practice has been very successful for this project.

3. **Frequent refactoring**
As expected, the initial tries of implementing classes and methods were very different from the final version of the developed applications. During the implementation, we have discovered several places where functionality could be reused, and where the structure of the system was about to grow too complex and difficult to follow. The refactoring functionality in IntelliJ IDEA has been invaluable and has enabled us to extract and rename methods, and move and rename classes without fear of breaking already working code. If this practice have not been used, it is very likely that we had lost control of the code structure and ended up with "spaghetti code" [9] already in the early phases of the project.

4. **Team code ownership**
In this project, we have looked at all written work as "ours" instead of "mine" or "his". Since we have been only two team members, this has been very important, as it has allowed both of us to quickly locate and fix errors in the work. If this had not been the case, such errors would have had to wait until the work's owner was available. This could quickly have led to much waiting and frustration for the team member not "owning" the work in question.

5. **Coding standards**
Following this practice was very simple in this project, as IntelliJ IDEA support automatic reformatting of the code on commits. Using this functionality, we were able to ensure that all committed code followed the agreed upon code standards. As previously explained, this simplified the practice of frequent refactoring, and understanding the code written by the other team member. Even though this was an easy practice to follow, it still proved very useful.

6. **Develop in short time-boxed iterations**
While XP operate with time boxes of only a few days per iteration, UP operate with longer time boxes of 2 - 6 weeks [34]. For this project, we have chosen an iteration length somewhere between these two. We selected an iteration length of one week, where we

started each iteration on Monday, and delivered working functionality from that iteration on Friday. Functionality we were not able to implement in that iteration was postponed to the next iteration. For this project, this iteration length provided us with flexibility enough that we were able to ensure the progress of the implementation, but still handle unpredicted problems. Furthermore, by not forcing the length of the iterations too short, we also had time to document the work being done, and work on this report.

7. **Develop the high-risk and high-value elements first**
   From our depth study, we already had a locally running client that worked satisfactory. The first priority was then to implement a simple server that was able to receive and forward data from connected clients. When this was completed, the remaining functionality was then implemented in a steadily increasing level of detail. Lower-value requirements were then easier to detect as a runnable system was always available. Combined with the "Simple design" practice, this approach helped us avoid the pitfalls of spending too much time on low-value elements and forgetting the more important ones.

8. **Cohesive architecture and reuse of existing components**
   To ensure a cohesive architecture, the system was implemented based on a combination of the Model-View-Controller and the Document-View architectural patterns, as described in Chapter 11. In this way, we always knew which components needed to communicate with which other components in both the server and the client application. Furthermore, because of the symmetry between the server and the client application, we could easily detect the functionality present on both sides. Finally, the extraction of functionality found to be common for all similar games into a framework eases the task of reusing this functionality for later projects within the same category. This practice was perhaps the most valuable of those we selected for this project, as it required us to continuously evaluate the code to ensure a logical structure of the system. Also, later developers using the code as basis for further development will very likely benefit from this choice.

9. **Ensure that you deliver value to your costumer**
   As mentioned, there are two parties that can be considered "costumers" in this project: IDI and Telenor. Identifying their goals for the project early on has been very valuable. It has enabled us to focus on the quality and contents of this report, as well as obtaining and evaluating our test results. If this had not been the case, there is a risk that far to much time would have been spent on trying to tweak the prototype game to perfection. Hence, having a continuous focus on following this practice has forced us to spend time obtaining a satisfactory quality on all parts of the project, where the costumers' priorities have been most important.

10. **Manage change**
    To keep track of change requests, we have used a text file where these requests have been entered along with a description of the request and where it should be performed. When a request has been met (or rejected), this has also been entered in the file. Even though this is a rather informal way of tracking change requests, it has worked very well for this project because of its limited size. Since the project team only consisted of two persons, keeping the change request list up to date has been an easy task using Subversion. Subversion has also been used to keep a history of the evolution of both the implementation and the project report. This way, we have had the possibility to compare versions and detecting the cause when errors have occurred seemingly with no apparent reason. Also, the use of Subversion has ensured that both team members at all times have worked on the latest version of the project, independent of the computer currently used.

The ten practices we have chosen to follow for this project have proved to be very useful and have assured the quality of the final delivery. The different practices have enabled a simple, flexible and effective development process, and have ensured a cohesive and logical structure of the system. Furthermore, the practices have helped with prioritizing the most important parts of the project, and disregard those with a lower value both for the project's results and "costumers".

On the negative side, we see that disregarding the practices involving continuous unit testing of the code may have led to some unnecessary effort with detecting and fixing bugs in the code. Still, as discussed, we believe that the effort involved with writing test cases for the different parts of the code would have required too much time given our previous experience with the subject. Even though use of test cases, when done correctly, may significantly increase the quality and stability of a system, we do not believe that we could have written these test cases detailed enough and still been able to both implement the prototype game and finish the report, given the short time scope of the project. Hence, we made a very good choice in our selection of practices to follow, and that these were very suitable for this kind of project.

Chapter 17

Technology Evaluation

The technologies used in this project were described in Chapter 5. In this chapter, these technologies are evaluated based on our experiences from this project. The technologies that are evaluated are the mobile network technologies and the transport protocols. These technologies have large influence on a real-time multiplayer mobile game's performance and playability. Each of the technologies is evaluated by their own properties and compared to the others. Through this comparison we will answer the questions regarding which mobile network technology and which transport protocol are best suited for a multiplayer mobile game.

17.1 Mobile Network Technologies

The mobile network technologies tested and evaluated in this project are listed and described in Section 5.3. The results from testing the network technologies were presented in Chapter 13. The evaluation of the mobile network technologies focuses on their ability to provide stable and fast enough data transfer between the clients and the server to ensure that the game has high playability and performance. The shorter response time and faster data transfer, the more real-time like the data updates will be. Thus, shorter response and transfer times mean higher playability and game performance. Taking cost issues into the evaluation is irrelevant, since with a Telenor subscription the cost is the same on all the mobile networks, except WLAN which is free.

17.1.1 GPRS

GPRS is the most widespread mobile network technology today. The long response and transfer times, however, show that GPRS is not ideal for playing a real-time multiplayer mobile game. With a response time around 500 ms, the real-time aspect is less functional and satisfactory than with a faster mobile network technology. The game flow suffers from this, and the graphical representation is more often flawed compared to other mobile network technologies. Also, GPRS' test results vary a great deal. This means that a stable response time can not be expected, which makes lag and delays more likely to occur.

17.1.2 EDGE

EDGE has better response and transfer times than GPRS in the tests. The response time of EDGE is low enough to make playing real-time multiplayer mobile games on this network satisfactory. Data updates are received from the server fast enough for the gameplay to be fun and exciting, but some delays may occur due to peaks in the response or transfer time.

17.1.3 UMTS

UMTS has the best response and transfer times of the widely available mobile network technologies. Playing a real-time multiplayer mobile game like BrickBlock with UMTS is more than satisfactory. Data updates are received fast enough for the game to have minimal synchronization issues or other problems. The UMTS test results vary less than GPRS and EDGE. Thus, lag and delay are less likely to occur with UMTS.

17.1.4 WLAN

WLAN has the best performance of the tested mobile network technologies. The response and transfer times are far shorter than UMTS, and as long as the mobile phone used is within the range of a WLAN network, the reliability of the network is also better than the other network technologies. The test results with WLAN is very consistent and coherent, i.e. the variation of the test results is minimal. The drawbacks of WLAN is its availability. Few mobile phones feature WLAN and users are dependent on being within range of a network that is open for use, or one they can log on to for a fee. However, when logged on to a WLAN, the data transfers are free. Another downside with WLAN is that the ports can be highly restricted, which means that the desired port number used in the game may not be available. Thus, the game may be unplayable with some WLANs. To solve this problem, the game must use a commonly open port number.

17.1.5 Conclusion

WLAN has the best performance, but because of its limited deployment and availability, UMTS, which is the best of the remaining tested mobile network technologies, provides the overall best services for a real-time multiplayer mobile game. The response and transfer times are short enough for the data updates to be received in such a time that the game flow is more than satisfactory and the gameplay entertaining and fun.

17.2 Transport Protocols

The transport protocols tested and evaluated in this project are listed and described in Section 5.4. The test results of the protocols are presented in Chapter 13. A real-time multiplayer mobile game demands fast and frequent data communication to ensure high synchronization and concurrent graphical representation on all clients. Therefore, the lower the response time and data transfer time the transport protocol provides, the better.

17.2.1 TCP

TCP is defined as a reliable connection-oriented transport protocol. This means that all sent data packets are guaranteed to be received in order. Packets that are lost are automatically retransmitted. However, to ensure this guarantee, the protocol uses a three-way handshake to acknowledge the connection. This slows down the response time and transfer speed of the protocol. On all the mobile network technologies we have tested, TCP has had longer response time and transfer time than UDP. Also, the transfer times vary more with TCP than with UDP.

The only mobile network technology that delivers satisfactory test results with TCP is WLAN. The response times with TCP on the most common mobile network technologies supported by mobile phones are too long to be satisfactory in a real-time multiplayer mobile game. In combination with a long transfer time, this makes this protocol unsuited for a such a game. Hence, TCP fits better to mobile games or applications that do not demand frequent and fast communication between a server and clients.

17.2.2 UDP

With UDP as transport protocol, packets (datagrams) are not guaranteed arrival and may arrive out of order. Though these properties can be considered drawbacks, UDP's smaller header and less overhead due to non-existent checking algorithms make UDP faster and more efficient than TCP. UDP has consistently shorter response times and transfer times in the test results on all combinations of mobile network technologies, send intervals, and packet sizes.

Even though more packets are lost with UDP than with TCP, UDP still provides the best transport protocol for a multiplayer mobile game. With faster response and transfer times, the data updates are received frequently enough that lost data packets are not noticed by the player, since the next packet arrives so fast. Losing a position packet is not so critical, since a movement prediction algorithm will mask the fault from the user and this event will not halt the game. However, since UDP does not guarantee delivery or retransmitting of lost packets, some lost critical data transmissions could hamper with the game flow. Important game commands from the server must be received by all the clients to ensure both the synchronization and the fairness of the game. Such important game commands could be the placement of power ups or other map objects. Despite the lack of guaranteed data packet receival, UDP is fast and efficient enough to be the best choice for a real-time multiplayer mobile game.

UDP also has the port number problem that WLAN has (Section 17.1.4). Some secure networks have specific rules for their ports and the allowed protocols. With these networks the port number must be defined as open for UDP for a server to be functional. Without this setup, data transmission will be blocked both to and from the specific port number. With less secure networks (for instance common household networks) such port number definition will not be necessary.

17.2.3 Conclusion

UDP is better suited for a real-time multiplayer mobile game than TCP. The game will gain more from fast transfer time and short response time than with guaranteed completion of sends since the data updates will occur so frequently. However, because of the need for confirmation of receival of critical data transmission and retransmission if a critical packet is lost, it would

be a good idea to implement an algorithm that fulfill this demand. This is not supported by UDP, but can be developed in the application using the protocol. Such an algorithm will take the best parts of TCP and add them to UDP, which would result in a fast and efficient transport protocol that can offer reliability and retransmission when needed.

Part V

Summary

Chapter 18

Answers to Research Questions

This chapter answers the research questions presented in Section 2.1.

18.1 Multiplayer Mobile Game's Challenges

1. Which challenges exist when developing real-time multiplayer client/server-games for mobile phones?

This research question was devised to identify the challenges related to developing real-time multiplayer mobile games so that future projects are able to work around or avoid problems. Most of these challenges are developer specific, and these were easy to deduce from our own development process. The other challenges are specific for the service provider or game host. These are mostly concerned with network and business issues.

18.1.1 Developer Challenges

(a) What are the developer specific challenges?

In Chapter 14, the problems encountered in this project while developing the prototype game are presented and discussed. Several of these problems were related to the different implementations of Java ME on the emulators and the test mobile phones. Ensuring that the application or game developed is functional on the desired mobile phone models is time-consuming and confusing work, since information about the mobile phones' Java ME implementation not always is publicly available. Thus, trying and testing is necessary. This problem can also result in some desired functionality not being available on all mobile phones models, thus leading to a limited customer base or a reduced version of the application or game.

Other challenges for the developer are creating a game with good gameplay and high entertainment value. A real-time multiplayer mobile game needs good game flow and frequent data updates to fulfill these demands. Ensuring the quality of the game flow and gameplay are done with good programming and algorithms. Player object movement, collision detection and handling, and game specific actions like pushing other players are key factors for the game flow and gameplay. Player object movement depends on position updates. The client receives position updates about the rest of the players from the server, and since the server sends these updates with set intervals, warping and lags may occur. To avoid these problems, movement prediction

algorithms may be implemented. The challenges with such movement prediction solutions are that they may cause additional false positions, increase the occurrences of warping, or make the control of collision detections difficult and inaccurate.

An additional challenge with the development of a multiplayer mobile game is the affect the frequency of data updates has on the cost of playing the game. With high data update frequency, the positions are updated more often, leading to more accurate presentations of the player objects on all clients and better playability. The occurrences of warping and other movement issues are decreased, leading to an improved game flow. The drawbacks of such frequent data updates are that more data is sent between the server and the clients. This means that the cost of the game increases, since the users pay for the amount of data downloaded. The developers must therefore either find clever solutions to reduce the size of each data transmission, maintaining the playability while decreasing the data update frequency, or a compromise that provides both satisfactory playability and cost.

18.1.2 Service Provider Challenges

(b) What are the service provider specific challenges?

The biggest challenges for the service provider are the technology and network aspects. For a multiplayer mobile game, the importance of a stable and available server is extensive. Without a stable server, the gameplay will suffer and players will not enjoy the game. Server stability means that peaks in latency or Round-Trip Time are avoided. This will ensure a fair game and better data updates. The stability of the server depends, among other things, on available data airtime, since speech and data transfer share the communication network. With an immensely popular game with a lot of players and extremely high amounts of data transfer in one specific area, the normal use of a mobile phone, phone conversations, may suffer, or vice versa. If the server is unavailable, i.e. the server goes down, playing the game will be not be possible. This will lead to unsatisfied players who have payed for a game without being able to play it. The server therefore has to be either so robust that it does not go down, or so easy and fast to repair or restart that the players have minimum downtime in their playing. The software issues with this are the developers' responsibilities, but the hardware issues are the service provider's. If the developers make a robust and available server application, the hardware must also be stable and solid.

Another big challenge is to develop a business model setup that is attractive to users. The game will not be a success if the price of playing is too high. One possible solution is a game subscription with a set price per month with included data transfer. With this model, the players can play the game without paying for the data transfer. Other solutions could be distributing the game as a free game or distributing the game as included software on mobile phones. Alternatively, the service provider could offer set prices for game specific data transfer (limiting the game time) or include the game and the data transfer in the ordinary mobile phone subscription at a set monthly price. This latter solution could be a good solution with an appealing and exclusive game to attract users to the mobile phone subscription. The business model must take into account both the service provider's and the game's users' interests.

In addition to cost, the service provider also must consider the aspects of distributing the game. One option is to bundle it as pre-installed software on a new mobile phone. This solution requires a collaboration with a mobile phone manufacturer. Such a collaboration may enhance the product (mobile phone) the manufacturer sells because it increases the amount of included software and entertainment value, resulting in more value for the costumer. Combined with an exclusive right for the specific game, this can be used as a marketing tool for selling

both a mobile phone and a subscription. Another option is making the game available for download from a WAP page. This makes the game available for all mobile phone users, and the game can be sold for a higher fee, since the user will be buying the game. With the first of these distribution options, a set price would have to be negotiated between the game developer and the mobile phone manufacturer. Since the number of games distributed like this will be very high, the price per game application will have to be lower. However, with a download distribution one never knows how many will buy the game. Thus, the bundle distribution option may be the safest alternative.

18.2 Mobile Network Technology

2. Do existing mobile network technologies for mobile phones provide satisfying properties for real-time multiplayer mobile phone games?

This research question was devised to determine if the currently available mobile network technology are suitable for a real-time multiplayer mobile game. Important aspects to consider here are the response time, the transfer speed, the overall playability with the specific network technology, and the cost involved with using the technology.

18.2.1 Response Time

(a) What are the differences between GPRS, EDGE, UMTS, and WLAN in terms of response time?

To answer this research question we developed and implemented the response time test module that is described in 12.1. This test module was used to test the different mobile network technologies. The tests results showed that UDP provides better response times than TCP, and that WLAN has the best response time with both transport protocols and with all send intervals. The average UMTS response time with UDP is approximately 200 ms longer than WLAN's average response time. EDGE's average response time is 150 ms longer than UMTS' average response time, and GPRS' average response time is 320 ms longer than the UMTS time. With TCP, the differences between the average response times are further increased. WLAN's average response time is approximately 650 ms shorter than UMTS', 750 ms shorter than EDGE's, and 1550 ms shorter than GPRS's. The results from the tests are further evaluated in Section 13.2.

18.2.2 Transfer Speed

(b) What are the differences between GPRS, EDGE, UMTS, and WLAN in terms of transfer speed?

Background for answering this research question was obtained by developing the transfer speed test module described in 12.2. Like the response time test module, this module was used to test the different network technologies and transport protocols. Again, the test results showed that UDP provides better results than TCP. UDP has the shortest transfer time on all network technologies. WLAN has the shortest transfer time independent of transport protocol with all packet sizes tested. WLAN's average transfer times with all packet sizes up to 760 bits are 7 ms and 32 ms on UDP and TCP, respectively. UMTS' average transfer times are 241 ms (UDP) and 753 ms (TCP), which is significantly higher than WLAN's. EDGE's average

transfer times are still higher, with 366 ms and 897 ms. GPRS' average transfer times are 492 ms and 2186 ms. The differences between the mobile network technologies' average transfer times are smaller with UDP than with TCP. Section 13.3 provide a more detailed evaluation of the transfer speed of the different network technologies.

18.2.3 Playability

(c) What are the differences between GPRS, EDGE, UMTS, and WLAN in terms of playability?

The playability of a multiplayer mobile game is determined by the entertainment value of the game. Accurate player representations and correct actions and action handling are key factors for a real-time game to be entertaining since these aspects lead to a game that functions as desired. The mobile network technologies' response and transfer times are the main aspects regarding the playability of such a game.

Because of WLAN's superior specifications, this network technology provides the best playability for a real-time multiplayer mobile game. With shorter response and transfer times, data updates are received faster and more frequently, thus warping and synchronization issues can be avoided. This also reduces the need for movement prediction. UMTS provides a playability that is satisfactory. The player movement is accurate enough for BrickBlock, and the gameplay is enjoyable and entertaining. However, because of the response and transfer times with UMTS, movement prediction must be used. This may lead to frequent faulty player object representation and warping. With GPRS, lags and warping happens more frequently than with the rest of the mobile network technologies, and the need for movement prediction is critical.

The test results and the playability on the other network technologies strongly indicate that EDGE's playability is a little worse than UMTS', but much better than GPRS' playability. EDGE's performance and movement prediction will provide satisfactory playability with less lag and warping than GPRS. When playing against opponents using different mobile network technologies, players may experience having an advantage when using a better technology than the opposition. Since this leads to an unfair and less entertaining game, dedicated servers to specific mobile network technology can be used to make sure all players have the same properties.

18.2.4 Cost

(d) How does the amount of data transfered affect the associated user cost when using the different transport protocols?

UDP is cheaper than TCP because of the smaller header. For each TCP packet transferred, 12 bytes more are sent than with UDP. The number of players in the game has the greatest affect for the cost. Other smaller influences on the cost are the player name length and the screen resolution, since these properties are sent from the clients to the server, and the server sends every player's info to every client. The number of actions performed by the player also influences the cost. The more a player tries to push a opponent, the more data is sent to the server. These cost issues are further discussed in Section 13.4.

18.3 Gameplay

3. What problems need to be solved to ensure a satisfying gameplay and how can they be solved?

The key to all entertaining games is a satisfying gameplay that makes users return to the game. A multiplayer game must be stable and enjoyable to play. Such a game must have minimum delays and inconsistency between clients to satisfy the users' demands.

18.3.1 Synchronization

(a) What methods can be used to ensure the game is sufficiently synchronized?

Complete synchronization of a multiplayer game running over mobile networks is impossible because of the latency involved with such networks. However, as described in Chapter 9, methods such as movement prediction, message bundling, and masking of delay can be used to synchronize or camouflage delays or differences to some degree. In our prototype game, movement prediction and message bundling are implemented. These two methods are used to avoid player warping and to increase the frequency of data updates on the clients.

The technique of movement prediction involves using previous known position data and movement directions to predict a remote player's position in the near future. In the framework, a simple movement prediction algorithm is used to give an impression that position updates are received more often than they really are. When the players move in straight lines and turn infrequently, this algorithm works satisfactory and gives an impression of a high degree of position exchange between the clients. However, if the players frequently change movement directions, this algorithm may lead to longer warping distances than not using it would have done.

The nature of the game decides what kind of movement prediction and masking of delay can be used. As discussed in Section 9.2, a game that uses gradual turning instead of immediate change of movement direction can use an interpolation method to predict the player's future position more accurate than our prediction algorithm. For example, car racing games could utilize from using this method.

Message bundling means to bundle many messages into to one, as described in Section 9.3. By bundling position messages to include all the players' positions when sending from the server, the number of messages that need to be sent are decreased, but the size of the position messages are increased. However, the message header, i.e. the information that states what kind of message it is, only needs to be sent once with this solution. This means that redundant data can be avoided. Furthermore, without bundled messages, the messages could end up in queues on the clients, since the clients will receive multiple messages for each send from the server. With message bundling, the generation and parsing time of the messages is increased. However, the improvement in data update frequency is high enough to justify using this technique.

A final method that could be used to synchronize the clients in a multiplayer mobile game is delaying the consequence of player actions until all clients have received notification of the action. For this purpose, animations can be used to hide the delay, so that from the user's point of view, the action appears to result in immediate consequences.

18.3.2 Connection Management

(b) What methods can be used to allow dynamic connections and disconnections?

Dynamic connections and disconnections means that a client can connect to and disconnect from a server without disturbing the other connected clients, which could result in reducing the player's feeling of a stable and well-flowing game. Firstly, it is clear that such a responsibility should lie with the server. The server has control over all clients currently connected, and if a client connects or disconnects, the server should be aware of this event. As long as connect and disconnect commands are received whenever a client connects to or disconnects from the server, this task is fairly simple. The server then only needs to add or remove the client from its list of connected clients, and notify all affected clients of the event. These clients then handle this event.

The problem with this connection management arises when the connection and disconnection commands are not received by the server. In a multiplayer mobile game this is a very likely event, since mobile networks can be unstable, and client applications may quit unexpectedly, for example if the mobile phone's battery runs out of power. Furthermore, when using UDP as transport protocol, packets may be lost on the way, and by the specification of the protocol, neither sender nor receiver are notified of this event.

Therefore, methods are needed to handle these events. In our implementation, if a connection request is lost, the server is never notified of this. Instead, the user of the client application is simply asked to retry the connection attempt. Since the client has not yet been connected to the server, this does not affect neither the server nor the other clients in any way.

However, if a disconnection request is lost, or not sent, the client will remain connected as far as the server and the other clients are concerned, even if this is not actually the case. This is solved by having the server send alive requests to all clients every two seconds. If a client fails to respond to several subsequent such requests, the server assumes that the client has disconnected, and notifies the other clients of the disconnection. For the other clients, this can then be handled as a regular disconnection. This method leads to disconnections being discovered a little late, but after some time, the disconnected client is removed, and the list of connected players will once again be correct both with the server and with the clients.

18.4 Multiplayer Mobile Game Framework

4. What requirements need to be fulfilled when developing a client/server multiplayer game framework?

By developing a game framework that can be reused in other multiplayer game projects, essential common functional requirements and functionality of such a framework are important to identify. Such requirements and functionality have little to do with the gameplay, but compose the system that is the backbone of a multiplayer mobile game. Examples of such backbone functionality is connecting to a server, creation of and connecting to sessions, lobby functionality, changing settings, keeping track of score, and player movement. The requirements identified as important multiplayer game framework requirements are described in Sections 10.1.1 and 10.2.1.

Non-functional requirements for a client/server multiplayer game framework are modifiability, availability, and performance. In Sections 10.1.2 and 10.2.2, the non-functional requirements used in this project are discussed. Since the framework is meant to be used in future multiplayer mobile game projects, modifiability is the key non-functional requirement. By ensuring that the framework has high modifiability, the framework is more likely to fit such projects. Adding functionality or modifying the framework will be faster and easier.

Also, high availability and performance will contribute to the ensuring the functionality of key elements for a successful multiplayer mobile game. High availability leads to less chance for the game to not function, resulting in that playing the game will be more entertaining and enjoyable. High performance provides better game flow and playability, since lags and delays are avoided and calculations are performed faster.

Chapter 19

Conclusion

In this master thesis, we have considered two separate, but still related, issues concerning real-time multiplayer mobile games. On one hand, we have performed tests to measure the suitability of today's available mobile networks for such games. On the other hand, we have developed a game framework and a prototype game, to evaluate the process of such development, and to determine whether such games can be played over the mobile networks widely available today.

When performing our tests of the different networks, we have discovered that the suitability of these networks for real-time games span from high to low. Also, we found great differences in the network performances dependent of the transport protocol used for the data transmission. For all networks, UDP provides far better performance than TCP. It is therefore clear that UDP should be used for this purpose, independent of the underlying network technology.

When connected to a WLAN network, real-time multiplayer mobile games can be played without problems, since this network technology provides both response times and transfer speeds that easily support such games. Furthermore, since the cost involved with using WLAN is independent of the amount of data transfer, a network game using this technology may send continuous data updates without considering the user cost involved with playing the game.

However, using WLAN as network technology suffers from low deployment. This applies both to the number of mobile phones supporting this technology, and to the number and range of publicly available WLAN networks nationwide. UMTS is therefore a far better suited technology when considering the deployment. Even though it does not have full coverage in Norway, most mobile users have access to a UMTS network. Our tests showed that this network technology also provides a performance sufficient for real-time multiplayer games, as long as UDP is used as the transport protocol. Unfortunately, the cost involved with using UMTS is currently not independent of the amount of data transfer as with WLAN. Instead, UMTS users are billed per MB of downloaded data. A real-time multiplayer game using UMTS therefore has to take this into consideration, and find ways to minimize the amount of data transmission, but still maintain a satisfactory game experience from the users' point of view.

The two last network technologies we considered were EDGE and GPRS. Of these, EDGE provides a performance not too far from UMTS, although a little more unstable. The properties of UMTS therefore also apply to EDGE. GPRS, on the other hand, is the only of these networks that is available anytime and anywhere. Unfortunately, the performance of GPRS networks is also considerably lower than the other networks, and far more unstable. Although GPRS

may be used for real-time multiplayer mobile games, it is therefore likely that the users of this network technology will experience frequent synchronization problems and an incoherent gameplay.

To solve the other part of this project, we developed a framework that can be used as a basis for developing real-time multiplayer mobile games, independent of the actual rules and gameplay of the game itself. To enable this, the framework contains functionality that we have determined to be common for all such games. This part of the project has been very successful, and the logical structure and basic functionality found in the framework provides a good basis and help future game developers get a kick-start with similar development projects.

Our prototype game is named BrickBlock, and is a direct extension of our framework. This game contains a very simple set of rules, and limited functionality beyond that already found in the framework. This has enabled us to evaluate the framework and to compare the findings from our tests with a concrete real-time multiplayer mobile game. Even though BrickBlock in itself is too simple and contains too little functionality to be attractive for mobile gamers, it has proved to be very useful in evaluating the possibility for successful games in this category.

The conclusion from testing and evaluating BrickBlock is that real-time multiplayer mobile games definitely have a chance of being successful. When using available network technologies that provide sufficient performance (UMTS and EDGE), the flow of the game can be taken to a level not much worse than that already known from computer based multiplayer games. However, both playing games on mobile phones and using such networks present challenges that are not present for computer games.

Firstly, mobile phones deliver far less performance than a computer does. This applies both to the computing power of the mobile phones' processor, the amount of available memory for a mobile phone, and the duration of the mobile phones' batteries. Because of these issues, developers of mobile phone games have to come up with clever solutions, that frees the mobile phone from having to perform too resource-demanding operations. For this purpose, a game server should be used to perform these kinds of operations, and only leave the mobile phones with the operations that have to be performed locally. However, this distribution has to be organized in a way that still keeps the amount of data transmission, and thus the user cost involved with playing the game, at a minimum. Alternatively, service providers may reduce the money cost of game data transmission, or offer set prices for game subscriptions. This will allow a high degree of data transmission and still keep the user cost low.

Furthermore, even though the mobile network technologies deliver good performance, this performance is not good enough for developers not to consider and work around network delays. Techniques have to be used that hide such delays from the users as much as possible, and give a feeling that the network performance is better than what is actually the case. Examples of such techniques are movement prediction and animations, where the first has been tested with mixed success in this project.

Finally, the degree of interaction between the players connected to a game has to be carefully considered. In BrickBlock, the players may push each other across the game board. This is an extreme degree of interaction, and has proved to be a significant challenge. We have implemented algorithms to try to make this work as good as possible, and have suggested techniques that might further improve the algorithms. Still, we are unsure whether this much interaction between the players is possible with today's available mobile network technologies. Developers of real-time multiplayer mobile games therefore have to carefully consider the degree of player interaction, and whether this is obtainable before defining a game concept. In many cases, a little less player interaction may be sufficient, and a lot easier to implement.

Chapter 20

Further Work

In this chapter we discuss possible further work aspects with this project. These aspects are suggestions to what may be added in the gameplay in projects aiming to improve the game. Also, possible work within the network aspect of the project is described.

20.1 Extending BrickBlock

As mentioned in previous chapters, the goal of the BrickBlock prototype game was never creating a fancy and advanced mobile game, but rather a very simple prototype game only containing the most necessary functionality. However, the current version of the BrickBlock prototype should be possible to extend with new functionality and features without too much effort. In this section, some ideas for how this could be done are presented.

20.1.1 Adding Bots

Since the game developed in this project is a multiplayer game, one will need adversaries to play a game. However, other players may not always be available, or a game could be more entertaining with more players than those currently available. Most multiplayer games are more fun with many players than with just a few. A possible way of solving this is adding "bots" to the game. Such bots are AI controlled players, or robotic computer controlled players. If such a solution is used, the offered gameplay would be expanded. The game could then include a singleplayer mode that could be used as a tool for helping new players understand the game, or for practicing the game.

If bots are supported, the players should be able to choose whether bots should be enabled or not, and the number of bots added to the game. In a singleplayer mode, the client could generate and control a limited amount of bots. This would reduce the amount of data having to be sent in the network. However, since more bots demand more resources, a mobile phone would only be able to handle a few bots with its limited resources. In a multiplayer mode with bots, the server would control the bots and send position updates to each client as normal.

The bots would move on the game board using pre-programmed routines. Such routines would have to take into account the trap position and the positions of power ups, as well as what decisions the bot should make when colliding with other players. The complexity of the AI

could range from low (easy bot) to high (hard bot). The aggressiveness (whether the bots "attacks" the other players or just tries to avoid the trap) of the bots could also be set. The goal of the AI would be to get the bot to make the same choices as a human player would, and act accordingly.

Bots could also be used for automatic testing of game sessions. By connecting multiple clients to the server with bots controlling each player object instead of a person, stress tests, data amounts tests, and availability tests could be performed automatic without the need of multiple human players. This would improve the test environment since the need for test persons would be very limited. Thus, project participants could spend their time with other project tasks, while tests are performed. Bot testing would be ideal for long and tedious, yet important, tests. In addition, bots can be used to demonstrate the game without the demonstrator playing the game.

20.1.2 Gameplay and Game Content

Since BrickBlock is a prototype game, its rules and contents are relatively simple. Functionality has been prioritized over advanced features. The game can easily be extended to have a more advanced gameplay. This could be done by adding additional power ups that would affect the players, or by adding objects that would change the game rules when a player picks it up. Examples of such game rule changes could be inverse player movement, or moving traps. Also, adding obstacles or additional traps would change the gameplay to some extent.

The graphics of the game are currently very simple. By using an explosion animation when players die, the game might feel more exciting and entertaining. Other graphical upgrades could be adding debris from a player collision or wall collision. This will make the collisions feel more violent and real, which would increase the entertainment value of the game. Furthermore, music and sound effects are currently not implemented in the game. These are normally important game parts that help set the feel of the game. Implementing music and sound in the game should be a simple task, as MIDP 2.0 includes basic audio capabilities through the Mobile Media API (MMAPI). This API is supported by most mobile phones supporting MIDP 2.0 through implementation of JSR-135.

The main concern when adding better graphics, sound and music, and additional gameplay aspects, is the mobile phones' resources. Added game content can not demand too much resources, so that mobile phones will have problems running the game. The calculation and data manipulation needed for better graphics and sound would be completed on the client since it would be difficult to synchronize it over the network. Also, the sound would be generated based on what the local player does, and the music might not have anything to do with the actions performed in the game. Players should be able to disable sound in case they are in locations where sound and music could be disturbing to other people. This would also limit the game's resource demands, so that mobile phones that have less available resources can disable the sound if needed.

20.1.3 Improving the Force Push Algorithm

In Chapter 14, we described the current problems with the force push algorithm. With its current implementation, it does not work satisfactory and needs improvement. Considering the rules of BrickBlock and the importance of force pushing each other, one can not say that BrickBlock runs satisfactory until this algorithm is improved. One of the first tasks for

a developer seeking to improve and extend BrickBlock should therefore be looking into the different solutions for this algorithm and finding a better suited implementation.

20.2 Extending the Framework

Even though the framework contains some functionality that we expect to be common for several different mobile multiplayer concepts, there is still a lot more functionality that could be included in the framework. This section presents some of the functionality that would further increase the value of the framework as a basis for future game concepts within this category.

20.2.1 Smooth Turning

As presented in Section 9.2, masking of delay and interpolation can be used to make the turning of the player objects more smooth than the 45 and 90 degree turns that are currently implemented in BrickBlock. These solutions would decrease the deviance between the predicted and the actual player position when movement prediction algorithms are used. However, it would also lead to more complex calculations and higher resource demands. In a game like BrickBlock, the current simple player movement can still be satisfactory because of the simplicity of the game. More advanced games such as a car racing game would most likely gain more from this type of player movement and movement prediction.

Support for smooth turning should therefore be implemented as part of the game framework. In this way, developers could decide which kind of turning is most useful for their kind of game, and select the appropriate method. Thus, the game framework would support a wider range of game types and time and development effort could be reduced for future multiplayer mobile game projects. As long as the movement algorithms are well documented and understandable, they would be useful for almost all multiplayer mobile games that feature player movement.

20.2.2 Confirmation of Critical Data Receival

As described in Section 5.4, a very important difference between using TCP and UDP as transport protocols is the reliability of TCP. When sending data over TCP, you can always be sure that the data sent will be received as long as the connection is maintained. However, as the results in Chapter 13 clearly show, UDP is better suited for real-time multiplayer mobile games than TCP because of its much better Round-Trip Time.

Therefore, an ideal transport protocol for this kind of games would be one that combines the speed of UDP with the reliability of TCP. In the `Communicator` interface, the `sendMessage()` method contains a flag for requesting confirmation that messages are received. In our current framework implementation, this flag is not used. Implementing a `Communicator` class that uses this flag and requests confirmation of the receival of critical data packets could significantly increase the stability of the framework. This would enable a developer to specify which data packets are (e.g. score), and which are not (e.g. position) critical for the execution of a game.

20.2.3 Other Networks and Protocols

New mobile network technologies are not far away from being available to the general mobile phone user. The most immediate of these networks were described in Section 5.3. By performing the same tests on these network technologies as the ones performed on the currently available technologies, one will get an impression of the improvements and see how much the game will benefit from the new technology in terms of performance.

This project has also only implemented support for two transport protocols. Other protocols such as SCTP (see Section 5.4.3), Real-time Transport Protocol (RTP), or GPRS Tunnelling Protocol (GTP) could be implemented. These implementations could be tested to see if they provide better performance than those tested in this project. However, the support for these protocols may be needed to be developed completely. In addition, not all mobile phones support other transport protocols than TCP and UDP.

20.3 Further Testing

In this project, several different tests have been performed to determine the performance of today's available mobile network technologies using both TCP and UDP as transport protocols. We have also tested the developed game prototype on a small test panel in an ad-hoc fashion to determine the playability of the game concept. However, even though our tests have been valuable tools in evaluating and answering our research questions for this project, there are several reasons for further testing of both our BrickBlock game and the available network technologies. This section provides a couple of examples for such tests.

20.3.1 Stability

In the testing, the stability of the game prototype have been tested by connecting and disconnecting to the server with both emulators and actual mobile phones. This allowed locating and improving flaws and weaknesses in both the framework and the BrickBlock prototype. However, we have not had the possibility to test the server in a real environment for longer periods of time. In such an environment, the server would be running for days and weeks without anyone operating the server. This could possibly lead to background processes in the server application gradually stealing the server's resources. Also, with such testing, errors might occur that have not been present during the limited stability tests performed.

In addition to the server running for a longer time period, a real environment would involve many clients in different locations continuously connecting and disconnecting from sessions. This leads to sessions being started and ended, and multiple simultaneous games being run to a much larger degree than have been tested. At some point, the server is likely to reach a limit where the processing power of the server can not handle the number of clients. Detecting this limit is important before such a game is deployed to ensure that the server's limits do not affect the connected players' experience of the game.

20.3.2 Usability Testing

As mentioned, the testing of BrickBlock's usability was performed in an ad-hoc fashion, where the reactions of the persons testing the game were observed and recorded . Since the priority

of BrickBlock was exploring the possibility of developing a real-time multiplayer mobile game, this method for testing usability was sufficient in this project. For a game designed to be distributed to buyers or subscribers, a more formal method for testing the usability should be used. In this way, different game designs can be evaluated, and the one best suited for deployment can be decided upon.

Chapter 21

Recommended Readings

This chapter gives a brief overview over books we have used throughout this project, and that we believe can be useful for other projects with a scope and purpose similar to this one.

21.1 Game Development with Java ME

The following books have been useful in this project because of their extensive presentation of game specific functionality with Java ME. In addition, these books helped us understand aspects within Java ME that are not only restricted to game development, but development in Java ME in general.

Wireless Java - Developing with J2ME [33] by Jonathan Knudsen describes how to program mobile phones, pagers, and other small devices using Java technology. The second edition of the book covers MIDP 2.0 and its new and enhanced features with special emphasis on MIDP 2.0's game API. This book was useful for us because we used plenty of MIDP 2.0's game related features and the book explains its subjects in great detail.

Beginning J2ME - From Novice to Professional [37] by Sing Li and Jonathan Knudsen is more concerned with Java ME in general than the other Knudsen book. We particularly found use for this book's very good introductions to network programming and persistent storage (RecordStores) in Java ME.

J2ME Games with MIDP2 [23] by Carol Hamer was helpful for us when we wanted to get started with developing using Java ME and MIDP 2.0. The book contains explanations for all the features in the MIDP 2.0's game API and emphasizes how to apply these features to games. It also describes network connections with Java ME, GUI aspects, other graphical elements, and the use of sound in Java ME games.

Micro Java Game Development [13] by David Fox and Roman Verhosek is a book about developing wireless games using Java ME. It is a step-by-step guide for creating games that gives an example of a wireless game.

J2ME in a Nutshell [56] by Kim Topley is a reference book for Java ME. It contains an introduction to the two different configurations in Java ME, CLDC and Connected Device Configuration (CDC), along with the MIDP profile and its APIs.

J2ME: The Complete Reference [**32**] by James Keogh is another reference book for Java ME, that helped us understand the organization, configurations, and profiles of Java ME. We also used this book to look up methods when needed.

21.2 Game Development in General

In addition to the Java ME game development books, we have used another book to gain knowledge and experience about developing multiplayer games in this project.

Massively Multiplayer Game Development [**2**] edited by Thor Alexander provides useful knowledge about technical solutions to many challenges that occur in development of multiplayer games.

21.3 Mobile Networks and Transport Protocols

Understanding the properties and workings of both mobile network technologies and transport protocols are important for every project concerning developing applications for mobile phones that are dependent on data transfer between mobile phones. The books listed in this section are suitable for gaining knowledge about the needed technologies for such projects.

GSM, GPRS and EDGE Performance [**22**] edited by Timo Halonen, Javier Romero, and Juan Melero gives an in-depth study of the performance and capabilities of GSM, GPRS, and EDGE, which is useful when performance comparing is the interest field. It also discusses different 3G technologies and how GPRS and EDGE are technologies that push the mobile networks toward 3G.

Mobile Radio Networks [**59**] by Bernhard H. Walke features comparisons of the performance of relevant mobile networks as well as discussions on next generation networks and WLANs. It can be used as a reference for mobile communications in general.

3G Mobile Networks [**31**] by Sumit Kasera and Nishit Narang helps with understanding the UMTS standard and answers questions around this technology. It explains the UMTS architecture, procedures, and protocols. It is well suited for any project utilizing UMTS.

TCP Performance over UMTS-HSDPA Systems [**3**] by Mohamad Assaad and Djamal Zeghlache presents overviews and analysis of the UMTS and HSDPA systems. It also describes the TCP protocol and its performance over HSDPA. The book covers problems that can occur with the use of TCP over wireless systems and solutions to such problems.

Computer Networking: Internet Protocols in Action [**38**] by Jeanna Matthews provides information about network protocols and compares the TCP and UDP transport protocols. The book discusses the strengths and weaknesses of both protocols, and presents results from various experiments.

Bibliography

[1] HSPA: High Speed Wireless Broadband - From HSDPA to HSUPA and Beyond. Technical report, UMTS Forum, 2005.

[2] Thor Alexander. *Massively Multiplayer Game Development*. Charles River Media, first edition, 2003.

[3] Mohamad Assaad and Djamal Zeghlache. *TCP Performance over UMTS-HSDPA Systems*. Auerbach Publications, 2007.

[4] Victor R. Basili. The Experimental Paradigm in Software Engineering. In *Lecture Notes in Computer Science*, pages 3–12. Springer-Verlag, 1992.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[6] James Belcher. Mobile Gaming is Taking Off, 2006. `http://www.imediaconnection.com/content/8022.asp`, last visited 2007-06-19.

[7] Fiona Chau. Mobile gaming aims for mass market, 2006. `http://www.telecomasia.net/article.php?id_article=1744`, last visited 2007-06-19.

[8] CipSoft. Tibia Micro Edition - the first mobile roleplaying game, 2007. `http://www.tibiame.com/home/?language=en`, last visited 2007-03-13.

[9] Community. Wikipedia, 2007. `http://www.wikipedia.org/`, last visited 2007-06-12.

[10] NTT DoCoMo. NTT DoCoMo, 2006. `http://www.nttdocomo.com/`, last visited 2007-06-19.

[11] Floodgate Entertainment. Floodgate Games, 2006. `http://www.floodg.com/`, last visited 2007-06-19.

[12] George H. Forman and John Zahorjan. The Challenges of Mobile Computing. Technical report, University of Washington, 1994.

[13] David Fox and Roman Verhovsek. *Micro Java Game Development*. Addison-Wesley, 2002.

[14] Bruce Gibson. Casual Gamers and Female Gamers to Drive Mobile Games Revenues Over the $ 10 Billion Mark by 2009, 2006. `http://www.juniperresearch.com/reports/23_mgames3/press_release.htm`, last visited 2007-01-04.

[15] Mitch Goldstein. *Hardcore JFC - Conquering The Swing Architecture*. Cambridge University Press, 2001.

[16] Network Working Group. RFC 768 User Datagram Protocol, 1980. `http://tools.ietf.org/html/rfc768`, last visited 2007-02-22.

[17] Network Working Group. RFC 793 Transmission Control Protocol, 1981. http://tools.ietf.org/html/rfc793, last visited 2007-02-22.

[18] Network Working Group. RFC 2960 Stream Control Transmission Protocol, 2000. http://tools.ietf.org/html/rfc2960, last visited 2007-02-22.

[19] Network Working Group. RFC 3286 An Introduction to the Stream Control Transmission Protocol, 2002. http://tools.ietf.org/html/rfc3286, last visited 2007-02-22.

[20] GSA. GSA - The Global mobile Suppliers Association, 2007. http://www.gsacom.com/news/statistics.php4, last visited 2007-02-08.

[21] GSM. GSM World - the website, 2007. http://www.gsmworld.com/technology/what.shtml, last visited 2007-03-05.

[22] Timo Halonen, Javier Romero, and Juan Melero. *GSM, GPRS and EDGE Performance - Evolution Towards 3G/UMTS*. Wiley, second edition, 2003.

[23] Carol Hamer. *J2ME Games with MIDP2*. Apress, 2004.

[24] Sumi Helal. Pervasive Java. *Pervasive Computing*, pages 82–85, January-March 2002.

[25] Sun Microsystems Inc. Java 6.0 API Documentation, 2006. http://java.sun.com/javase/6/docs/api/, last visited 2007-06-06.

[26] Sun Microsystems Inc. Sun Java Wireless Toolkit 2.5.1 for CLDC Download , 2007. http://java.sun.com/products/sjwtoolkit/download-2_5_1.html, last visited 2007-06-06.

[27] Sun Microsystems Inc. and Motorola Inc. MIDP 2.0 API Documentation, 2006. http://java.sun.com/javame/reference/apis/jsr118/, last visited 2007-06-06.

[28] Jan Krikke. Samurai Romanesque, J2ME, and the Battle for Mobile Cyberspace. *IEEE Computer Graphics and Applications*, 2003.

[29] Martin Jarrett and Eivind Sorteberg. Proximity Based Multiplayer Games For Mobile Phones. 2006. Depth study at IDI at NTNU.

[30] Kalle Jegers and Mikael Wiberg. Pervasive Gaming in the Everyday World. 2006. Umeå University.

[31] Sumit Kasera and Nishit Narang. *3G Mobile Networks*. McGraw-Hill, 2005.

[32] James Keogh. *J2ME: The Complete Reference*. Osborne, 2003.

[33] Jonathan Knudsen. *Wireless Java - Developing with J2ME*. Apress, second edition, 2003.

[34] Craig Larman. *Agile and Iterative Development - A Manager's Guide*. Addison-Wesley, 2004.

[35] Neal Leavitt. Will Wireless Gaming Be a Winner? *Computer Magazine*, pages 24 – 27, January 2003.

[36] William C. Y. Lee. *Wireless & Cellular Telecommunications*. McGraw-Hill, third edition, 2006.

[37] Sing Li and Jonathan Knudsen. *Beginning J2ME - From Novice to Professional*. Apress, third edition, 2005.

[38] Jeanna Matthews. *Computer Networking: Internet Protocols in Action*. Wiley, first edition, 2005.

[39] mDisney Studios. mDisney Studios, 2006. http://mobile.disney.go.com/, last visited 2007-02-10.

[40] Dirk Michel and Nathan Ramasarma. GPRS Measurement Methodologies and Performance Characterization for the Railway Environment. 2005. Wireless Communications and Network Conference, http://ieeexplore.ieee.org/iel5/9744/30730/01424764.pdf, last visited 2007-02-09.

[41] MiKTeX. MiKTeX project website, 2006. http://www.miktex.org/, last visited 2007-04-12.

[42] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical report, HP Laboratories Palo Alto, 2003.

[43] Tommi Pelkonen. Mobile Games, E-Content Report 3. In *Anticipating Content Technology Need (ACTeN 2004)*, pages 1–25, February 2004.

[44] Mike Abolins Pocketgamer.co.uk. Mobile platforms explained, 2007. http://www.pocketgamer.co.uk/r/Mobile/feature.asp?c=1266, last visited 2007-05-01.

[45] Michael Powers. Mobile Multiplayer Gaming, Part 1: Real-Time Constraints, 2006. http://developers.sun.com/techtopics/mobility/midp/articles/gamepart1/, last visited 2007-02-28.

[46] Rahul Raje. Extreme Programming (XP). Technical report, Illinois Institute of Technology, 2003.

[47] Karl Rottmann. *Matematisk Formelsamling*. Spektrum forlag, sixth edition, 2001.

[48] Peter Van Roy and Seif Haridi. *Concetps, Technologies, and Models of Computer Programming*. MIT Press, first edition, 2004.

[49] Thomas Senneset. Transparent Adaptable Network Access and Service Content Differentiation. Master's thesis, ITEM at NTNU, 2006.

[50] Sun. Mobile Devices with Java Support, 2006. http://wireless.java.sun.com/devices, last visited 2006-12-17.

[51] Sun. Java SE 6 Features and Enhancements, 2007. http://java.sun.com/javase/6/webnotes/features.html, last visited 2007-01-26.

[52] Telenor. Telenors Årsrapport 2004: Årsberetning, 2004. http://www.telenor.no/rapporter/2004/arsberetning/, last visited 2007-02-09.

[53] Telenor. Telenor - Priser - GPRS, EDGE og UMTS, 2007. http://telenormobil.no/priser/tjenester/gprs/, last visited 2007-03-14.

[54] Telenor. Telenor: Welcome to Telenor, 2007. http://www.telenor.com, last visited 2007-01-31.

[55] ToolsCenter.org. ToolsCenter.org, 2006. http://texniccenter.sourceforge.net/front_content.php, last visited 2007-06-19.

[56] Kim Topley. *J2ME In a Nutshell*. O'Reilly, 2002.

[57] Odd R. Valmot. Mobilt internett med fart. *Teknisk Ukeblad*, 16:26–27, 2007.

[58] Hans Van Vliet. *Software Engineering - Principle and Practice*. Wiley, second edition, 2002.

[59] Bernhard H. Walke. *Mobile Radio Networks*. Wiley, second edition, 2002.

[60] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying Ye. *Probability & Statistics for Engineers & Scientists*. Prentice Hall, seventh edition, 2002.

[61] Alf Inge Wang. *Using a Mobile, Agent-based Environment to support Cooperative Software Processes*. PhD thesis, IDI at NTNU, 2001.

[62] Alf Inge Wang, Michael Sars Norum, and Carl-Henrik Wolf Lund. Issues related to Development of Wireless Peer-to-Peer Games in J2ME. In *First Conference on Entertainment Systems (ENSYS 2006)*, pages 1–6, Guadeloupe, French Caribbean, February 23-25 2006.

[63] Christopher Williams and Mark Burge. MIDP 2.0 Changing the Face of J2ME Gaming. Technical report, Sun Microsystem, 2004.

[64] Verizon Wireless. Mobile Games Details from Verizon Wireless, 2006. `http://getitnow.vzwshop.com/search_games.aspx?id=search_games&appSearchParentCategoryId=247&appSearchText=pirates`, last visited 2007-05-01.

[65] Verizon Wireless. Mobile Games from Verizon Wireless, 2006. `http://getitnow.vzwshop.com/index.aspx?id=games&bhcp=1`, last visited 2007-06-19.

[66] Verizon Wireless. Verizon Wireless V CAST Technology, 2006. `http://getitnow.vzwshop.com/index.aspx?id=vcast_technology`, last visited 2007-06-19.

Part VI

Appendices

Appendix A

Glossary

This chapter contains the abbreviations and acronyms used throughout our project. To the left of each line, the abbreviation is found, while the full name is found on the right.

1G First-Generation Technology

2G Second-Generation Technology

2.5G 2.5-Generation Technology

2.75G 2.75-Generation Technology

3G Third-Generation Technology

3GPP Third Generation Partnership Project

4G Fourth-Generation Technology

8-PSK Octagonal Phase Shift Key

ADSL Asymmetric Digital Subscriber Line

AI Artificial Intelligence

API Application Programming Interface

BREW Binary Runtime Environment for Wireless

CDC Connected Device Configuration

CDMA Code Division Multiple Access

CLDC Connected Limited Device Configuration

CS Coding Scheme

DCCP Datagram Congestion Control Protocol

DS-CDMA Direct-Sequence CDMA

EBNF Extended Backus-Naur Form

ECSD Enhanced Circuit-Switch Data

EDGE Enhanced Data rates for GSM Evolution

EGPRS Enhanced General Packet Radio Services

EV-DO Evolution-Data Optimized

FDD Frequency Division Duplex

FOMA Freedom of Mobile Multimedia Access

FPS Frames Per Second

GMSK Gaussian Minimum Shift Key

GPRS General Packet Radio Services

GSA Global mobile Suppliers Association

GSM Global System for Mobile Communications

GTP GPRS Tunnelling Protocol

GUI Graphical User Interface

HS-DSCH High-Speed Downlink Shared Channel

HSDPA High-Speed Downlink Packet Access

HSPA High-Speed Packet Access

HSUPA High-Speed Uplink Packet Access

i-Appli Internet Applications

IDE Integrated Development Environment

IDI Department of Computer and Information Science

IEEE Institute of Electrical and Electronics Engineers

IMP Information Module Profile

IP Internet Protocol

ITU International Telecommunication Union

Java EE Java Platform, Enterprise Edition

Java ME Java Platform, Micro Edition

Java SE Java Platform, Standard Edition

JAD Java Application Descriptor

JAM Java Application Manager

JAR Java Archive

JDK Java Development Kit

JRE Java Runtime Environment

JSR Java Specification Request

JVM Java Virtual Machine

JWT Java Wireless Toolkit

KVM Kilobyte Virtual Machine

LAN Local Area Network

MAN Metropolitan Area Network

MIDP Mobile Information Device Profile

MMAPI Mobile Media API

MMMG Massively Multiplayer Mobile Game

MMO Massively Multiplayer Online

MMORPG Massively Multiplayer Online Role Playing Game

MMS Multimedia Messaging Service

MVC Model-View-Controller

NTNU Norwegian University of Science and Technology

OTA Over-The-Air

P2P Peer-to-Peer

PAN Personal Area Network

PDA Personal Digital Assistant

PDF Portable Document Format

PTM Point-To-Multipoint

PTP Point-To-Point

PTT Push-To-Talk

RPG Role-Playing Game

RTP Real-time Transport Protocol

RTT Round-Trip Time

RUP Rational Unified Process

SCTP Stream Control Transmission Protocol

SMS Short Message Service

SVN Subversion

TD-CDMA Time Division CDMA

TDMA Time Division Multiple Access

TCP Transport Control Protocol

TDD Time Division Duplex

TSN Transmission Sequence Number

TTI Transmit Time Interval

UDP User Datagram Protocol

UML Unified Modeling Language

UMTS Universal Mobile Telecommunications System

UP Unified Process

UTRA UMTS Terrestrial Radio Access

VM Virtual Machine

W-CDMA Wideband CDMA

WAP Wireless Application Protocol

WiFi Wireless Fidelity

WLAN Wireless LAN

WMAN Wireless MAN

WPAN Wireless PAN

WYSIWYG What You See Is What You Get

XML Extensible Markup Language

XP eXtreme Programming

Appendix B

Running BrickBlock

This chapter provides descriptions of how to install the BrickBlock server application on a computer, and the client application on a mobile phone.

B.1 Running the BrickBlock Client

This section explains how to install and run BrickBlock on a mobile phone. The phone models we *know* are compatible with BrickBlock are those listed in Table 5.1. However, we have not experienced that BrickBlock has failed to run on any phones supporting MIDP 2.0.

Installing the client

To run the client application on a mobile phone, the files called `bbClient.jad` and `bb-Client.jar` must be installed on the phone. This can be done in several ways.

- If both `bbClient.jad` and `bbClient.jar` are located on a web server supporting WAP, the application can be installed using WAP. Enter the address to the jad-file (e.g. <serveraddress>/bbClient.jad) in the phone's WAP browser, and the application will be installed automatically.
- If the phone supports wireless communication with a computer, for example via Bluetooth, the jar-file can be transferred from the computer to the mobile phone. When the jar-file has been transferred, select the `bbClient.jar` file using the mobile phone, and the application will be installed.
- If the computer containing the jar-file has a memory card reader, and the phone has a memory card, the jar-file can be copied to the memory card using the computer. Installing the application will then be similar to the previous process.

Starting the client

Starting the BrickBlock client application should be very easy when the application has been installed. Simply navigate the phone to where the application was installed (usually in a

'Games' folder), and select the application. The startup screen should then be displayed, and the address to a running server can be entered.

B.2 Running the BrickBlock server

In this section, a short introduction to how to install and run the BrickBlock server application on a computer with an Internet connection is provided. The methods for running the server spans from very easy to a little more complex. However, installing the server is very simple.

Installing the server

To install the server, simply store the file called `bbServer.jar` in the local filesystem on the computer where the server shall be run.

Starting the server

Since the server is implemented in Java 6.0, it requires Java Runtime Environment (JRE) 6.0 (or later) to be able to run. This can be downloaded and installed from `http://java.sun.com/javase/downloads/index.jsp`.

When JRE 6.0 is installed, the server can be started in several different ways. What method should be used depends on the desired port number and transport protocol, and whether or not run performance tests are to be performed.

The easiest way to run the server is double clicking the `bbServer.jar` file in a file browser, like Windows Explorer. This starts a standard server that uses port number 15 000 and UDP for its communication. However, if the server is started this way, no server status messages, such as player connections and disconnections will be printed to the screen. Only the server GUI will be visible.

If the user wants to see server status messages while the server is running, other methods must be used. These methods are described in the following listings. All of these methods require that the user opens a console window[1] and navigates to where the jar-file is located[2]. For the last three procedures found in the listings, the values annotated '<...>' must be exchanged with the valid values shown in Table B.1 found at the end of this section.

Listing B.1 shows the procedure for starting a server with the default values (port number 15 000 and UDP as transport protocol). This procedure corresponds to the first method, but using this procedure, server information will be printed in the console window. Thus, information on connections, disconnections, and started and stopped games is provided.

Listing B.1: Running server with default values

```
1 java -jar bbServer.jar
```

To specify the port number and transport protocol for the server, the procedure shown in Listing B.2 must be used.

[1]To open a console window in Microsoft Windows: Start -> Run -> Type "cmd" -> OK.
[2]Type "cd <path to the directory of the jar-file>" and press enter.

Listing B.2: Running server with specified values

```
java -jar bbServer.jar <port number> <communication protocol>
```

When running the test modules, the transport protocol in use has high significance. Both the test type and the transport protocol must therefore be specified. Listing B.3 shows the procedure for running a test using the default port (15 000).

Listing B.3: Running test with default port number

```
java -jar bbServer.jar <test type> <communication protocol>
```

The final method for running the BrickBlock server is a test run where both server port and communication protocol are specified. This is shown in Listing B.4.

Listing B.4: Running test with specified port number

```
java -jar bbServer.jar <port number> <communication protocol> <test type>
```

`<port number>`	An integer in the range *[0, 65 535]*
`<communication protocol>`	*udp* or *tcp*
`<test type>`	*ping* or *speed*

Table B.1: Valid arguments when starting a server

Appendix C

Extended Backus-Naur Form

Extended Backus-Naur Form is one of the most common notations for defining grammars [48]. The notation distinguishes terminal (a token) and nonterminal symbols (a sequence of tokens). Nonterminal symbols are defined by a grammar rule that shows how to expand the symbol into tokens. Table C.1 shows the notation of the EBNF. The last notation in the table is the notation used in this project to represent comments on a grammar.

Table C.1: EBNF notation

<table>
<tr><th>Notation</th><th>Definition</th></tr>
<tr><td><...></td><td>Symbol</td></tr>
<tr><td>::=</td><td>Definition</td></tr>
<tr><td>|</td><td>Choice</td></tr>
<tr><td>[...]</td><td>Option</td></tr>
<tr><td>{...}</td><td>Repetition</td></tr>
<tr><td>?...?</td><td>Special sequence</td></tr>
</table>

To read such a grammar, one starts with any nonterminal symbol and reads the corresponding rule from left to right. Each terminal symbol is added to the sequence, and each nonterminal is replaced by the sequence of tokens that it expands into. Whenever there is a choice, any of the alternatives are selected. If there is an option, the symbols are added to the sequence zero or one times. Finally, if there is a repetition, the symbols are added to the sequence any number of times (including zero).

Listing C.1: An example of grammar rules defined with EBNF

```
<digit>         ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
<int>           ::= <digit> { <digit> } ;
<number>        ::= <int> [ <decimalsymbol> <int> ] ;
<decimalsymbol> ::= "." ;
```

The first line in Listing C.1 defines the terminal symbol <digit> as a representation of one of the ten digits from 0 to 9. The "|" is read as an "or", meaning that one of the alternatives has to be selected. The second line defines the nonterminal symbol <int> as a <digit>, followed by any number of digits because of the braces ({...}). The nonterminal symbol <number> consists of an <int> followed by zero or one <decimalsymbol> <int> sequences. The brackets

([...]) means that the content should be added one or zero times. The `<decimalsymbol>` is the terminal symbol for the decimal separator used in this grammar.

Appendix D

Detailed Architecture

In Chapter 11, the architecture of the game framework and its BrickBlock implementation were described. This appendix contains a more detailed description of this architecture. In particular, the communication between the clients and the server are explained in detail. Also, a description of how future developers can use the game framework to implement their own real-time multiplayer mobile games is provided. Finally, more detailed class diagrams than the high-level diagrams provided in Figure 11.4 and Figure 11.5 are included in the last section of this appendix.

D.1 BrickBlock Messages

In Section 11.2.2, the actions and messages used in the framework and the BrickBlock prototype game were described and specified. This section contains a more detailed description of the contents of these action messages. In addition, at the end of this section examples messages are provided, to illustrate how the messages are composed. Further descriptions of how and when these messages are transmitted between the parties are provided in Section D.2.

D.1.1 Message Specifications

In this section, the messages following each of the actions specified in Table 11.1 and Table 11.2 are further specified. These specifications follow a syntax loosely based on the EBNF notation described in Appendix C. However, instead of specifying each non-terminal symbol, the left side of the specifications contains the name of each action. In addition, the low-level symbols are not further specified, since the contents of these are self-explanatory.

Listing D.1 shows the specification of action messages sent from a client to the server during a game session. The first actions are implemented in and supported by the framework, whereas the last two actions are BrickBlock specific, and hence implemented in the BrickBlock implementation of the framework.

Listing D.1: Client ⇒ server message specification

```
PLAYER            : "PLR:"<name>","<red>","<green>","<blue>","<score>","<team>","<readystate>
                    ","<positionX>","<positionY>","<resolutionX>","<resolutionY>"|"
ALIVE_REQUEST     : "ARQ:"<requestId>"|"
SESSION_SELECTED  : "SES:"<sessionId>"|"
```

```
SETTING_CHANGED   : "SET:"<name>","<value>{";"<name>","<value>}"|"
READY             : "RDY:"<state>"|"
START             : "STA:|"
DISCONNECT        : "DIS:|"
POSITION          : "POS:"<positionX>","<positionY>"|"
SCORE             : "SCR:|"

POWERUP_REMOVED   : "PUR:"<powerupType>","<positionX>","<positionY>"|"
FORCE             : "FRC:"<victimId>","<movementX>","<movementY>"|"
```

In Listing D.2, the action messages sent from the server to one or more clients are specified. Like in the previous listing, the first actions are contained in and supported by the framework, whereas the last five actions are BrickBlock specific. The reason why the server ⇒ client messages have more BrickBlock specific messages is that these messages are triggered by server events, and thereby generated only by the server. As can be seen, the most important difference between the messages specified in Listing D.1 and Listing D.2, is that some of the latter contain a player id field ended by a ”@” character. This is to specify the player to which the action applies.

Listing D.2: Server ⇒ client message specification

```
PLAYER            : "PLR:"<playerid>"@"<playerName>","<red>","<green>","<blue>
                          ","<score>","<team>","<readystate>","<positionX>","<positionY>
                          ","<resolutionX>","<resolutionY>"|"
ALIVE_REQUEST     : "ARQ:"<requestId>"|"
SESSION_LIST      : "SSL:"[<sessionId>","<sessionName>","<nofPlayers>","<maxPlayers>
                          {";"<sessionId>","<sessionName>","<nofPlayers>","<maxPlayers>}]"|"
SETTING_LIST      : "STL:"<settingName>","<value>{";"<settingName>","<value>}"|"
SETTING_CHANGED   : "SET:"<playerId>"@"<name>","<value>{";"<name>","<value>}"|"
READY             : "RDY:"<playerId>"@"<state>|
START             : "STA:"<resolutionX>","<resolutionY>","<positionX>","<positionY>"|"
DISCONNECT        : "DIS:"<playerId>"|"
POSITION          : "POS:"<playerId>","<positionX>","<positionY>
                          {";"<playerid>","<positionX>","<positionY>}"|"
SCORE             : "SCR:"<playerid>"@"<score>","<positionX>","<positionY>"|"
GAME_OVER         : "GAM:"<reason>"|"

TRAP_ADDED        : "TRA:"<positionX>","<positionY>"|"
POWERUP_ADDED     : "PUA:"<powerupType>","<positionX>","<positionY>","<increment>"|"
POWERUP_REMOVED   : "PUR:"[<playerId>"@"]<powerupType>","<positionX>","<positionY>
                          ","<increment>"|"
POWERUP_INACTIVE  : "PIN:"<playerId>"@"<powerupType>","<positionX>","<positionY>
                          ","<attributeValue>"|"
FORCE             : "FRC:"<pusherId>"@"<victimId>","<movementX>","<movementY>"|"
```

D.1.2 Example Messages

This section contains one example of each of the action messages specified in the previous section. Naturally, the values shown in the examples will be different from game to game, and from message to message. Only the actions (the first three letters of the message) and the action separator (the ”:”) will always be as shown in the examples.

Listing D.3 shows example action messages sent from a client to the server during a game session. This listing correspond directly to Listing D.1 provided in the previous section.

Listing D.3: Examples of messages sent from client to server

```
PLAYER            : PLR:1337plr,224,33,59,0,-1,false,-1,-1,240,291|
ALIVE_REQUEST     : ARQ:53|
SESSION_SELECTED  : SES:2|
SETTING_CHANGED   : SET:Score limit,8;Teams enabled,true|
READY             : RDY:true|
START             : STA:|
DISCONNECT        : DIS:|
```

```
POSITION          : POS:58,167|
SCORE             : SCR:|

FORCE             : FRC:2,-2,0|
POWERUP_REMOVED   : PUR:SPD,120,89,2|
```

Example messages for the opposite communication direction, server ⇒ client, are shown in Listing D.4. These example messages correspond directly to those specified in Listing D.2.

Listing D.4: Examples of messages sent from server to client

```
PLAYER            : PLR:3@1337plr,224,33,59,0,-1,false,-1,-1,240,291|
ALIVE_REQUEST     : ARQ:53|
SESSION_LIST      : SSL:0,Session #0;1,Another session;3,Third|
SETTING_LIST      : STL:Score limit,5;Time limit,0;Teams enabled,false;Players,8|
SETTING_CHANGED   : SET:Score limit,8;Teams enabled,true|
READY             : RDY:3@true|
START             : STA:240,291,89,76|
DISCONNECT        : DIS:4|
POSITION          : POS:1,230,150;2,0,0;3,58,167|
SCORE             : SCR:3@-3,230,281|
GAME_OVER         : GAM:1337plr lost the game!|

TRAP_ADDED        : TRA:120,88|
POWERUP_ADDED     : PUA:SPD,120,89,3|
POWERUP_REMOVED   : PUR:3@SPD,120,89,3|
POWERUP_INACTIVE  : PIN:3@SPD,120,89,1|
FORCE             : FRC:3@2,-2,0|
```

D.2 Sequence Diagrams

This section provides five sequence diagrams that show how the actions described in Table 11.1 and Table 11.2 are send forth and back between the server and clients during a game session. In most of the diagrams, only one of the clients is shown as an "active" client to simplify the diagrams. All actions performed by this client could also be performed by any of the other clients.

In the diagrams, three different labels are used. The messages sent between server and client are all written with capitalized letters. These messages all correspond to the defined actions, and the contents will be as specified in Section D.1. Commands that are selected by the local user are annotated with '<...>', whereas events detected by the applications (server or client) are annotated with '[...]'.

D.2.1 Joining an Active Session

Figure D.1 shows the initial steps of a session. Clients 2 and 3 in the figure are already connected to the session before Client 1 connects. The following list explain the three steps illustrated in the figure.

1. The player tries to connect to a server with the attributes specified in the startup screen. His attributes are sent to the server, but the client does not receive a notification that the connection was successful.
2. The player selects to refresh the session list to retry connecting to the server. His attributes are once again sent to the server, and this time the server responds by returning a list of the active sessions.

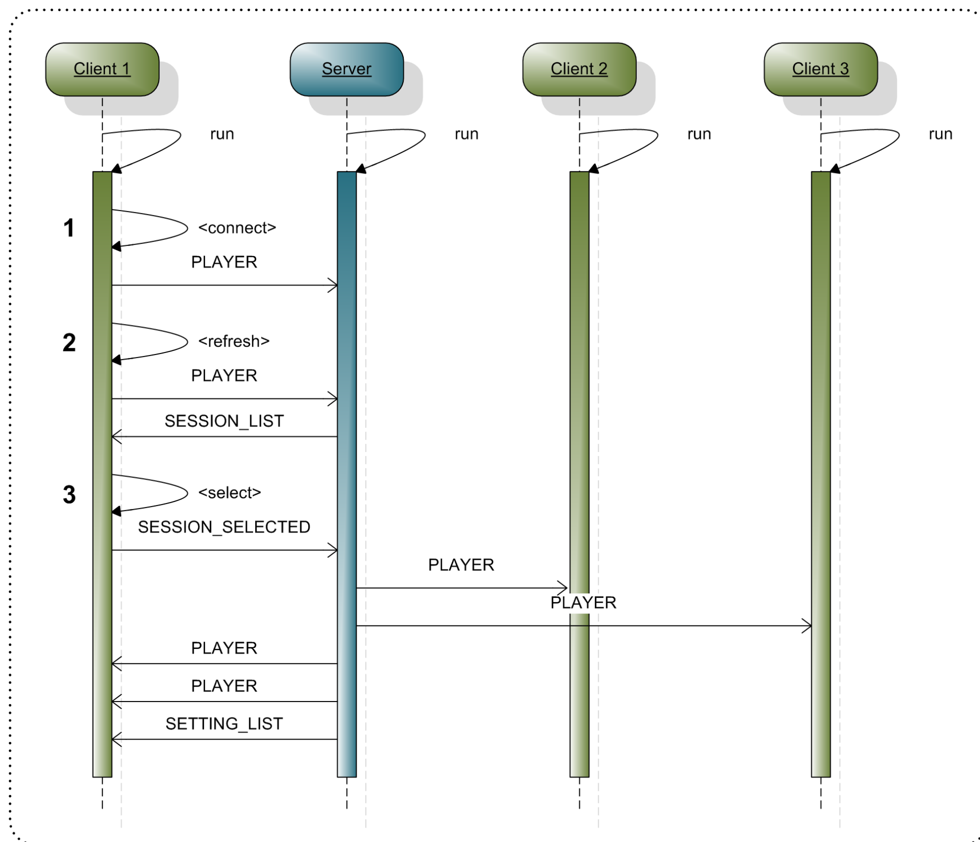

Figure D.1: Joining an active session

3. The session containing the other two players is selected, and the other two clients are notified that a new player has connected to the session. Finally, the new client receives player information about the other two players, and a list containing the active settings for the session.

D.2.2 Interacting in the Lobby

In Figure D.2, the communication between players waiting in the lobby is illustrated. The four steps shown in the figure are described in the following list.

1. The player wants to see the active settings for the session, and selects the 'Settings' command. These settings have already been received from the server, and are displayed in the settings window. The player then changes the value of one or more settings, and selects the 'OK' command. The changed settings are sent to the server, and immediately forwarded to the other connected clients.

2. The player feels ready for a game. The change of status is then sent to the server, and forwarded to the other clients.

3. Since the other players have also signaled that they are ready for a game, the player decides to start a game. The start request is sent to the server, who in turn notify all

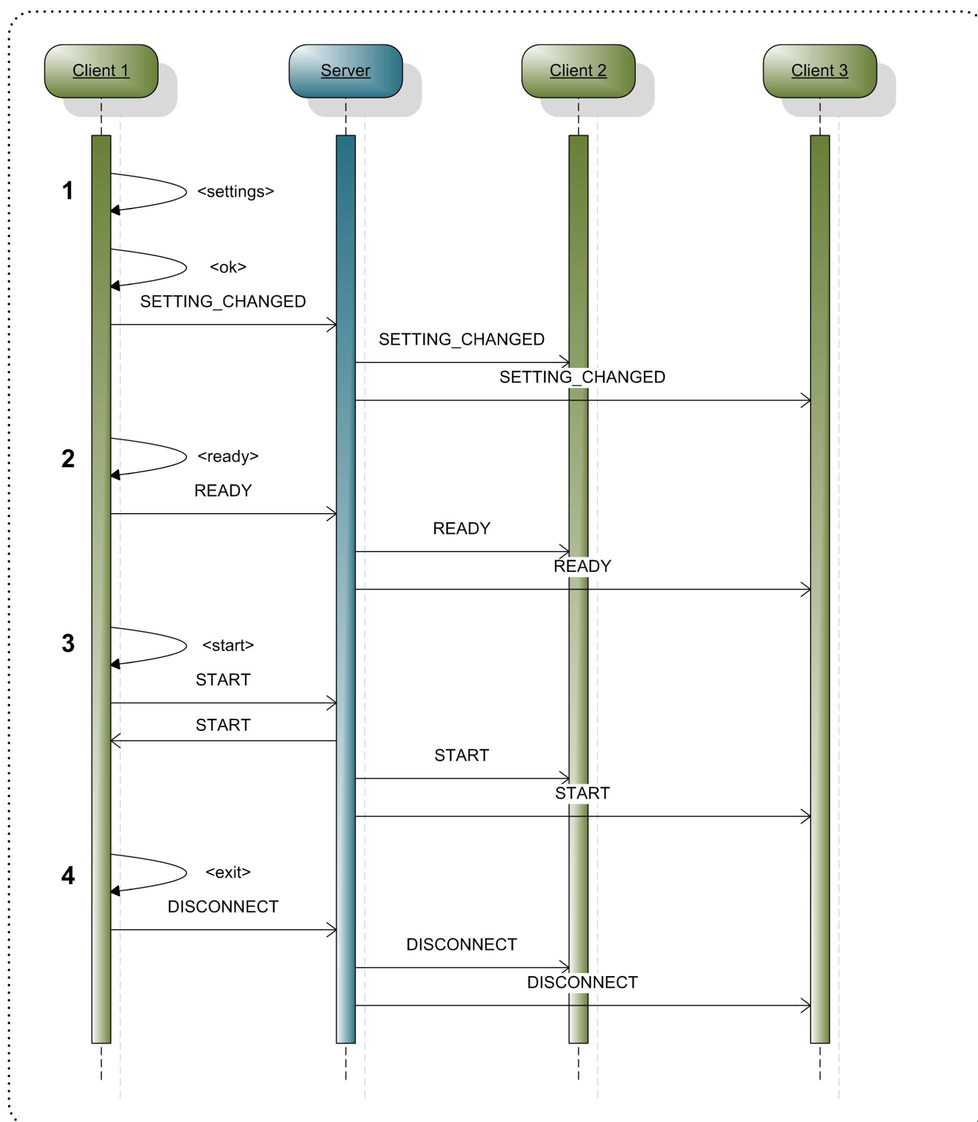

Figure D.2: Interacting in the lobby

clients (including the one initiating the game) that a game is about to start.

4. The player is tired of playing, and decides to leave the game. He then selects the 'Exit' command, and a disconnect command is issued to the server before the client application is closed. The server then notifies the other clients about the disconnection.

D.2.3 Playing a Game

The sequence diagram shown in Figure D.3 shows the sequence of message transmission triggered by a client. The steps shown in this figure are repeated continuously throughout a game, and can occur in any order. Each of the four steps are explained in the following list.

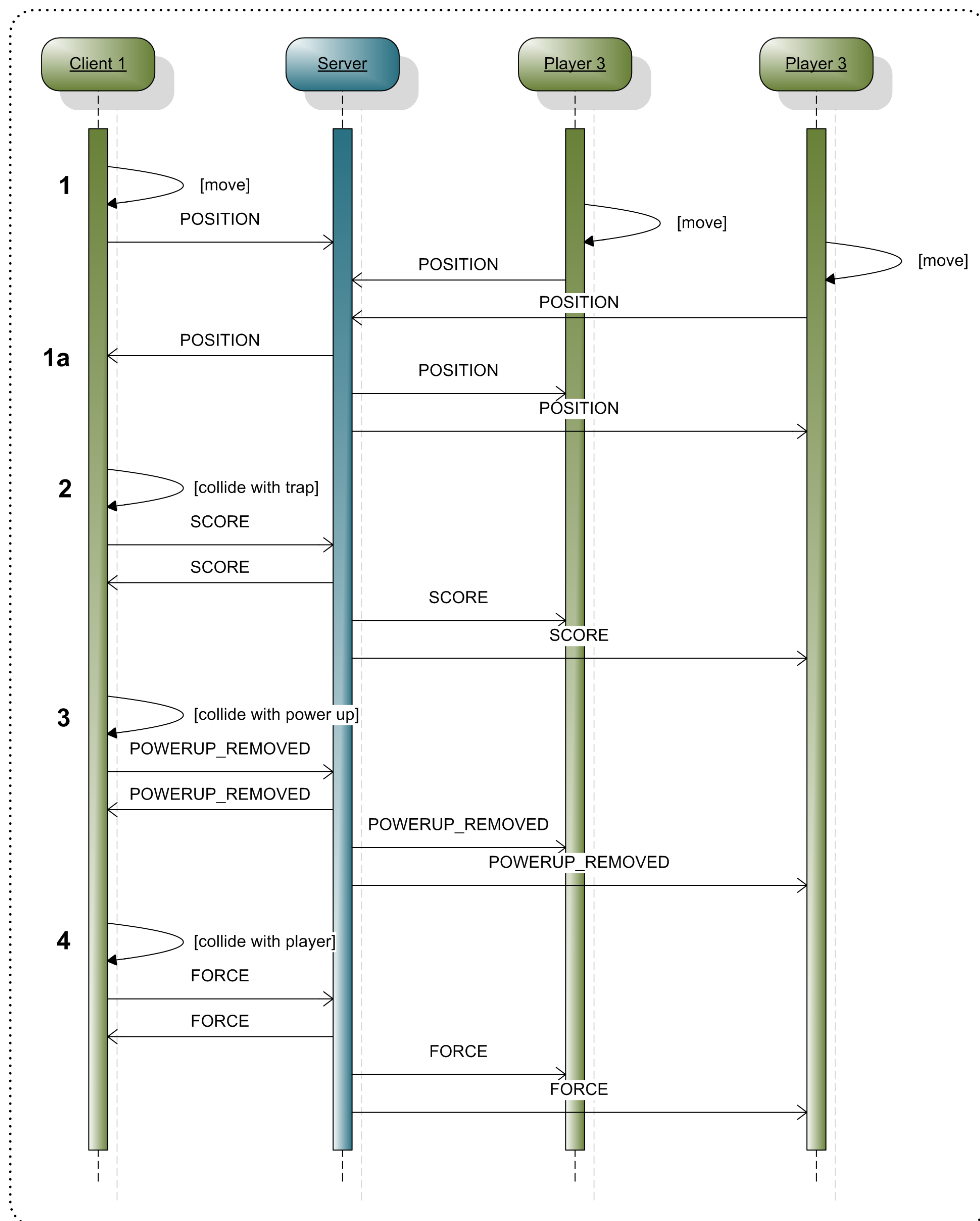

Figure D.3: Client game sequence

1. The player moves his object by pressing the defined movement keys. When the client's `PositionThread` detects that a move has occurred, the player's new position is sent to the server. The same happens for the other two connected clients. The server receives these position updates, for the time being without forwarding the messages.

   (a) The `GameThread` on the server has decided that it is time to transmit the position updates it has received. These updates are then bundled into one single position message and transmitted to all connected clients, who then update their game boards accordingly.

2. The BBBoard has detected that the local player has collided with the trap. A notification of this event is transmitted to the server. The server then generates a resurrection position for the unlucky player, and sends a notification that the player has received a negative point and has been moved to a new position.
3. The BBBoard has detected that the local player has collided with a power up object. The server is notified of this event, and determines if the power up object is still available. If so, a notification that the power up has been picked up is transmitted to all clients (including the one whose player picked up the power up).
4. The final game event occurs when the local player collides with another player. This is detected by the framework's GameView class. When this happens, a force vector is sent to the server, and forwarded to all connected clients (including the one that sent the force command).

D.2.4 Administrating a Game

In Figure D.4, a sequence diagram that shows messages triggered by server events is provided. Like the sequences in Figure D.3, these steps are also repeated continuously throughout a game, except the first and the last step. Also, the three middle steps may occur in any order. The steps shown in the figure are explained in the following list.

1. When a new game is started, the server generates a trap position, and transmits this position to all connected clients.
2. With irregular intervals, the server generates random power up objects. When such an object is generated, the power up's attributes are sent to the connected clients.
3. If a power up is not picked up by any of the players before its predefined duration is up, the clients are notified that they need to remove the power up object from the game board.
4. Step 3 in Figure D.3 showed the procedure when a power up is picked up. After a predefined span of time, the power up will no longer be active for the player that picked it up. This is detected by the server. When this happens, all clients are notified about the event. The specified player's attributes are then reset to their previous values on all clients.
5. Continuously throughout the game, the server compares the game's state to the session's settings. When a setting limit is reached, all clients are notified that the game has ended. The clients then remove the game board and display the lobby view.

D.2.5 Detecting Disconnections

The server is responsible for detecting when a client has been disconnected from the server, but the disconnection notification has not been received. This is done by sending alive requests with regular intervals, and removing clients that fail to respond to several subsequent such requests. The sequence diagram in Figure D.5 shows the steps of this procedure. These steps are explained in the following list.

1. The player connects to the server.
2. The server sends two (or more) alive requests, and the client responds to both.

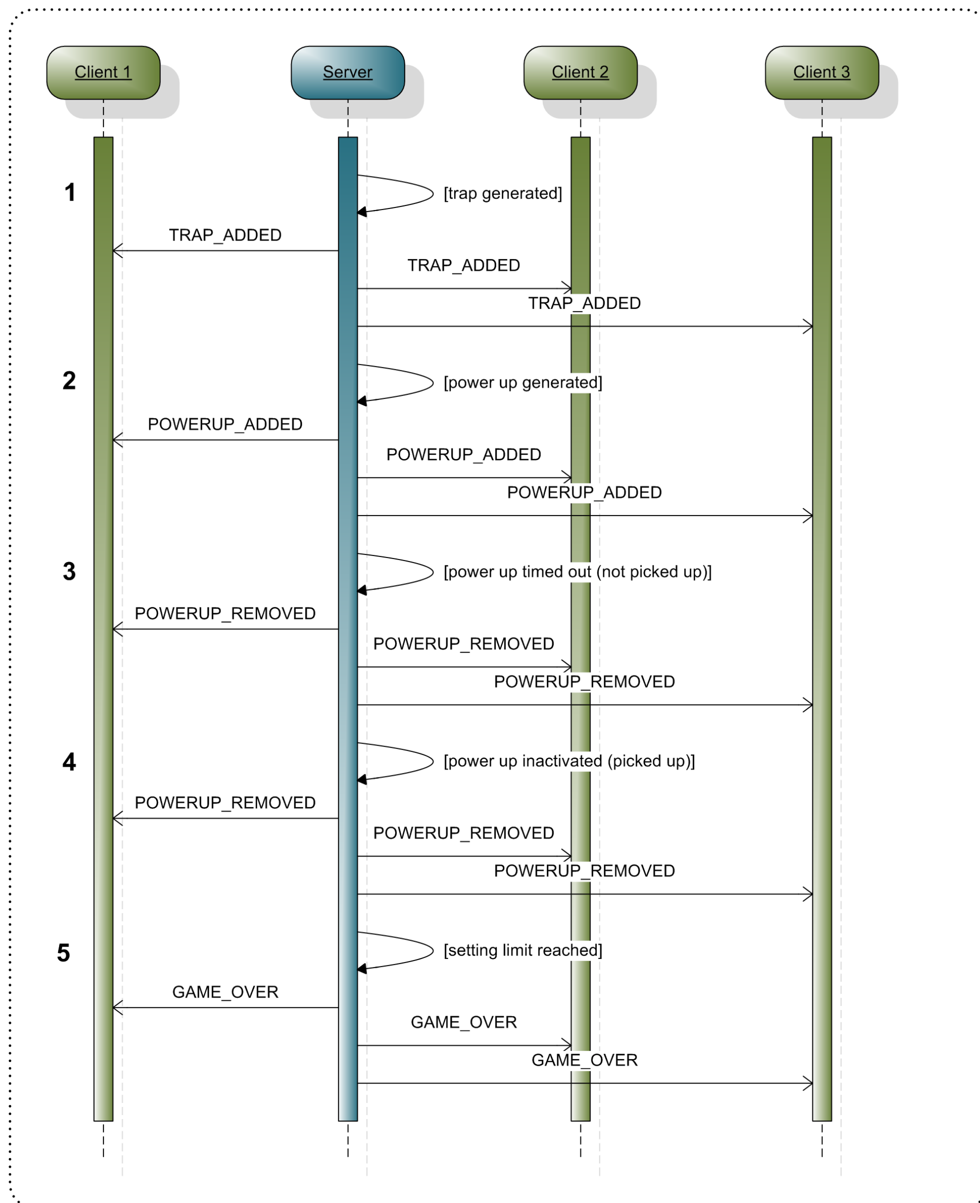

Figure D.4: Server game sequence

3. The server sends three more alive requests, and the client fails to respond to all these requests.

4. If the lost requests limit is three, the event described in the last step results in the server deciding that the client must have been disconnected. Thus, the client is removed from the server's player lists, and all other clients are notified about the event.

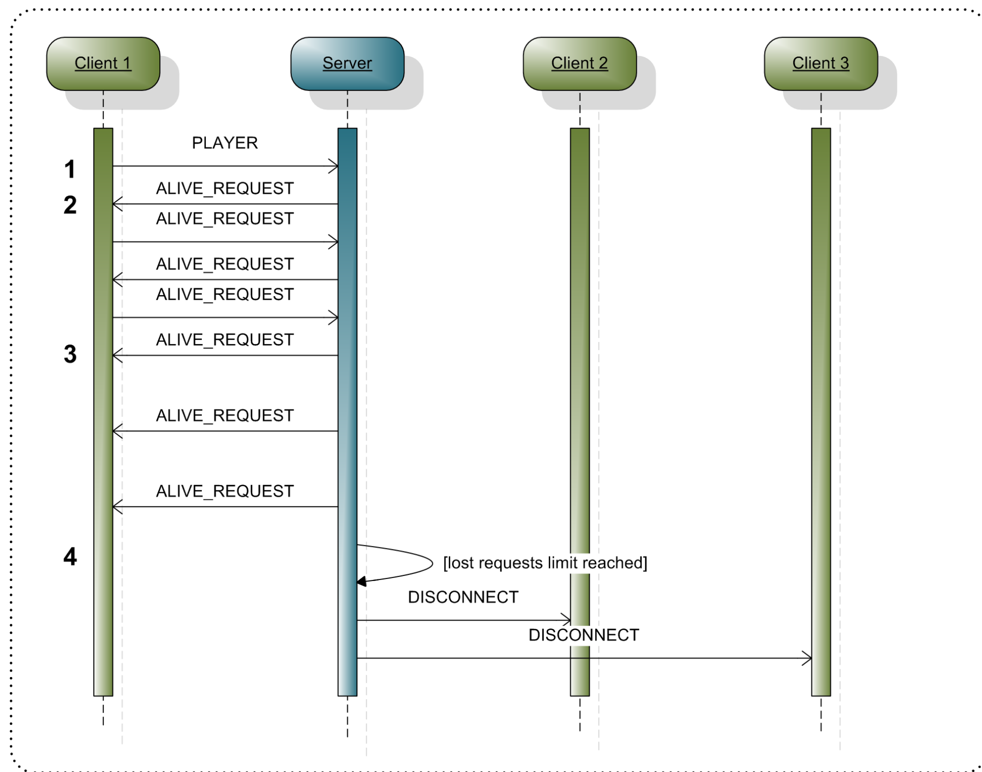

Figure D.5: Alive requests

D.3 Extending the Framework

This section gives an introduction to how to extend the framework to implement a real-time multiplayer mobile game. On both the server and the client, a number of abstract classes *must* be implemented in order for the applications to compile without errors. In addition, some classes should be extended or have contents added if new functionality need to be implemented. The first section gives an overview of these classes for the client application, whereas the last section gives an overview of the server's classes.

D.3.1 Client

For the client framework, there are three classes that need extension, and one that may be extended to add functionality. These classes, and their most important methods are explained subsequently.

The `AbstractGame` class (`framework.models`)

The `AbstractGame` model was described in Section 11.3.1. As mentioned, this model is the client's most important model, and keeps track of the client's other models. In addition, the `AbstractGame` model communicates with the client's communication module. The methods that need implementation in classes extending this class are described in the following list.

`createGameView()` returns a `GameView` object of the type specified in the framework implementation. This method is simply used to create the right type of game view object.

`getGameName()` returns the name of the framework implementation. In our implementation, this method returns "BrickBlock".

`newPlayer(String address, String name)` returns an `AbstractPlayer` object of type specified in the framework implementation. This method is similar to the `createGameView()` method, and is used to create the right type of player object.

`notifyAboutSpecialActionReceived(AbstractPlayer player, String action, String[][] values)` is called if a message is received that can not be interpreted by the framework's standard functionality. When new actions are added to a game, these actions need to be handled in this method's implementation.

`handleSpecialSettings(boolean init)` is called to see if any of the game's settings should be detected by the client to stop a game in progress. Since the server handles most settings, this method is not likely to have any content.

The `AbstractPlayer` class (`framework.models`)

The `AbstractPlayer` model is the representation of the players connected to the session. This model keeps track of the player's game object, score and other player related information. This method does not contain any abstract methods. Thus, the `AbstractPlayer class` contains enough functionality to represent simple player objects, and an extension of this class actually does not need any content. However, once functionality is added to the game, it is likely that this class needs to be extended with additional functionality.

The `GameView` class (`framework.views`)

As described in Section 11.4.1, the `GameView` implementation is responsible for showing the game board when running a game, using the contents of the `AbstractGame` model implementation. Since the look and functionality of a game is very dependent of the game's rules, this class contains a number of abstract methods that needs to be implemented. These methods are explained in the following list.

`createBackgroundImage()` creates an `Image` object that is used as the background for the game board. In the BrickBlock implementation, this method creates an image that fades from yellow to white and back to yellow.

`specialCheck()` is the method that is responsible for detecting game events other than collisions. This method is called each time the game thread loops, i.e. several times per second. In the BrickBlock implementation, there are no such extra events, and the contents of the method is therefore empty.

`handlePlayerCollision(AbstractPlayer collidesWith, int[] movement)` is called whenever a collision with another player is detected. The other player and the movement that causes the collision is provided as input parameters, and this method's responsibility is handling this event according to the game rules. When this method is called, the move has not yet been performed. If the move is allowed, the method therefore needs to move the player according to the given movement vector.

`detectObjectCollision(AbstractPlayer player, int[] movement)` is called to detect collisions with game objects. Since no game objects other than player objects are specified in the framework, this method needs to be implemented in the game implementation. If an object collision is detected, the following method should be notified and handle the collision.

`handleObjectCollision(Object collidesWith, int[] movement)` works similarly to the `handlePlayerCollision()` method, but is called when object collisions are detected.

`clean()` is called when a game is over. This method's responsibility is clearing all object lists, thus resetting the game board before the next game.

The `Action` class (`framework.network.parser`)

The `Action class` is not an abstract class, but extending this class is still likely to be necessary if extra functionality is added to the game. The default contents of this class are the actions specified in Table 11.1. Thus, extra functionality can be added by creating a subclass of this class. In the BrickBlock extension, this is done by creating the `BBAction class`.

D.3.2 Server

The server framework contains four abstract classes that must be implemented, one more than the client framework. However, most of these extensions are only concerned with specifying the type of objects used in the different classes. Since the server is implemented in Java version 6.0, and generics are supported, this can be specified in the class declaration. This reduces the need for casting objects in the code, which may lead to exceptions if the casting is done incorrectly. The different classes that must be extended, and those that may need extensions, are explained subsequently.

The `AbstractServer<AbstractSession, AbstractPlayer>` class (`framework.models`)

The `AbstractServer` is, as explained in Section 11.3.2, the top level model for the server application. The model handles creation of new player objects, and receives notifications from the server's communication objects when messages are received. The `AbstractServer class` contains three abstract methods that must be implemented when extending this class.

`createPlayerInstance(SocketWrapper wrapper, TestModule.TestType testType)` creates and returns a new player object of the type specified in the class' declaration. This method is called when a new client connects to the server.

`createSessionInstance(TestModule.TestType testtype)` is similar to the previous method, but instead creates and returns a new session when a connected player has signaled that he wants to create a new session.

`notifyAboutSpecialMessageReceived(ActionPair actionPair, P sender, S session)` is called whenever a communication object has received an action that is not contained in the framework's specified actions. It is then up to the implementation of this method to determine what this action means, and act accordingly.

The `AbstractSession<AbstractPlayer>` class (`framework.models`)

This model does not need to be extended with much functionality, since thread handling is already implemented in the framework, and the `AbstractServer` model handles communication with the server's communication objects. The two abstract methods in this class are explained in the following list.

`createEventHandler()` creates and returns the `EventHandler` implementation used in this framework implementation.

`notifyAboutEventOccured(Event event, P sender, Object[] values)` is called whenever the server model receives a messages that leads to an event occuring. The implementation of this method can be done in many ways, but in BrickBlock, the event is simply forwarded to the `EventHandler` thread.

The `AbstractPlayer` class (`framework.models`)

This model corresponds to the `AbstractModel` in the client framework. The difference between these is that this model in addition to the player's attributes also contain the communication object assigned to that particular player. Assigning this communication object is done automatically by the framework. The only required content in extensions of this model is therefore a constructor taking the same input parameters as the default class' constructor.

The `EventHandler<AbstractPlayer, AbstractSession>` interface (`framework.threads`)

The `EventHandler` is not an abstract class, but an interface extending the `Runnable` interface. This interface specifies methods that illustrates the thread's responsibilities. The following list explain the methods specified for this interface.

`setSession(S session)` sets the session to which the thread belongs. The session must be of the type specified in the class' declaration.

`fireEventOccured(P player, Event event, Object[] values)` is called whenever the thread's owning session detects that an event has occurred, that the thread needs to handle. This typically happens when the server receives a game related message.

`checkSettings()` is called continously throughout a game. This method runs through the session's settings list, and should detect if any setting limits are reached. The thread is responsible for calling this method itself.

`stop()` is called when a game is over. The thread should then empty all lists and free all occupied resources, to make these available for other processes.

`isReady()` is called to ensure that the thread has performed all necessary calculations before a game is started. Examples of such calculations are player and trap position generations.

The `Event` enum (`framework.threads`)

This enum only contains a list of the different events handled by the `EventHandler` implementation. By default, the framework only contains score and join events (the join event is not used). Since enums can not be extended, developers adding support for other events need to add these events directly to the `Event` enum.

The `Setting` enum (`framework.models`)

The `Setting` enum contains a list of the settings that are used in the game implementation. Each of these settings has a name, which is the name shown in the settings list in the client application. Similar to the Event enum, this enum also needs to have content added directly if support for new settings is added to the game.

The `SettingsList` class (`framework.models`)

This class contains a list of the settings that are used in the session containing the `SettingsList` object, and their values. When new settings are added to the `Setting` enum, these settings also need to be added to the `SettingsList` class, along with a default value. This should be done in the `initialiseSettings()` method.

D.4 Class Diagrams

Due to the size of the architecture's class diagrams, these are not included in the report. They would have been unreadable if they had been included. Therefore, the diagrams are provided in an own folder with the project's attached files. The class diagrams are located in the folder named *Class Diagrams*. They are stored in the .png image format. Each package has its own class diagram, and the class' package is displayed in the diagram's top left corner.

Appendix E

Files

This appendix summarizes the attached files. Each folder's contents are briefly explained in their own sections. Further subfolders' content are also explained.

E.1 Applications

This folder contains the runnable applications developed in this project.

E.1.1 Client

This subfolder contains the `bbClient.jad` and `bbClient.jar` files described in Section B.1.

E.1.2 Server

This subfolder contains the `bbServer.jar` file described in Section B.2.

E.2 Class Diagrams

This folder contains the class diagrams for the applications' architecture.

E.2.1 Client

This sub-folder contains the class diagrams for the client application.

E.2.2 Server

This subfolder contains the class diagrams for the server application

E.3 Javadoc

This folder contains the javadoc documentation for both the server and client implementation. To read the javadoc, open the `index.html` file.

E.4 Source Code

This folder contains the source code for the applications.

E.4.1 Client

This subfolder contains the source code for the client application.

E.4.2 Server

This subfolder contains the source code for the server application.

E.5 Test Results

This folder contains the test results from the tests performed in this project.

E.5.1 Response Time

This subfolder contains the response time test results.

TCP

This subfolder contains the response time test results when using TCP as the transport protocol.

UDP

This subfolder contains the response time test results when using UDP as the transport protocol.

E.5.2 Transfer Speed

This subfolder contains the transfer speed test results.

TCP

This subfolder contains the transfer speed test results when using TCP as the transport protocol.

UDP

This subfolder contains the transfer speed test results when using UDP as the transport protocol.

E.5.3 Large Data Amounts

This subfolder contains the test results from the large data amount tests.