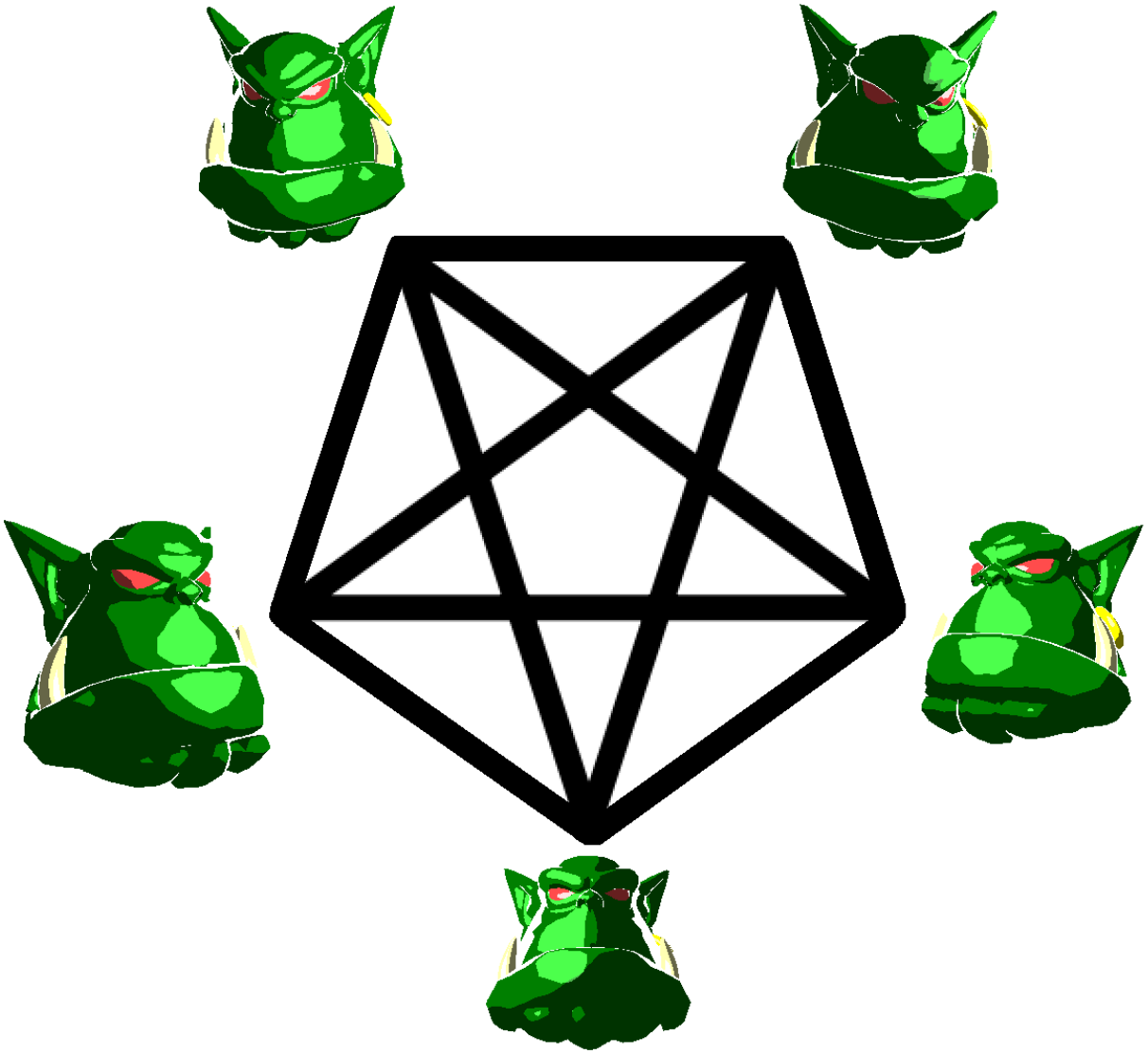
RawRlocks

A Fast-Paced Peer-to-Peer Game



Master Thesis by

Janus Hansen, Rune Kristian Jensen & Martin Breum Rosenbeck

Title:
Rawrlocks – A Fast-Paced Peer-to-Peer Game

Theme:
Distributed Systems

Project timeframe:
1^{st} February - 31^{th} May, 2011

Project group:
f11d602a

Group members:
Janus Hansen
Rune Kristian Jensen
Martin Breum Rosenbeck

Supervisor:
Brian Nielsen

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg
Telephone:(45)96358080
http://www.cs.aau.dk

Abstract:

Real-time multiplayer action games require fast communication between computers. The most common solution is to use a central server for communication, but other solutions are possible. Some games designate one of the players to be the server. Another approach is a peer-to-peer solution where every peer in the game is part server and part client. This solution has not been proven successful in any commercial games, but may be a viable solution.
In order to test a peer-to-peer solution, we create a simple fast-paced action game called Rawrlocks. The game is used to examine the peer-to-peer architecture and evaluate if that architecture is feasible for a real-time multiplayer action game. Maintaining consistency in Rawrlocks is split into two problems. 1) variables that can be modified by all players should only be accessible by one player at any time. To solve this problem we implemented a synchronisation service. 2) how to ensure that events are executed at the same time across peers. We solve this by implementing a functionality that delays all events for an equal amount of time.
Our testing shows that variables managed by the synchronisation service are only modified by one peer at any time. Also, most events are executed on both peers within a time frame of 3ms as long as the latency does not exceed the amount of time the events are delayed by.

Copies: 5

Total pages: 88

Report finished: 30^{th} May, 2011

Signatures

Janus Hansen

Rune Kristian Jensen

Martin Breum Rosenbeck

Preface

This report is written during the Dat6 project period by computer science group f11d602a on the 10th semester at Aalborg University. The main theme of this report is "Distributed and Embedded Systems". This report is addressed at students, supervisors or anyone who finds the topic interesting. However it is required to have a knowledge base equivalent to that of a 10th semester computer science student.

References to sources are marked by [#], where # refers to the related literature in the bibliography at the end of the report. References with the format [# P. #] refer to a specific page in the literature. Figures without a reference have been made by the group.

In the back of the report is a CD containing:

- The "Game" folder from our repository containing the source code of Rawrlocks and game necessary files.
- The "program" folder from our repository containing source code for the lobby server, lobby client and synchronisation service and the scripts used to automate the tests.
- A folder with the parsers used to parse test data.
- All the logs gathered from the tests.
- Aggregated data from the tests.

Contents

1 Introduction

Ever since the first multiplayer game *Tennis For Two* emerged in 1958, there has been focus on multiplayer in games. Before the Internet became popular it was often via modem or locally players played with each other. Nowadays, most multiplayer gaming is happening via the Internet, and some makes a living out of competitive gaming. There are two main genres of multiplayer games: Games that are played in a persistent world and games that are not played in a persistent world. Games in a persistent world are called MMOG's (massively multiplayer online game) and games that are not played in a persistent world are called session based games. An example of an MMOG is *World of Warcraft* and a session based game example is *Counter Strike*. Multiplayer adds a social aspect to computer games where you test your skills against friends or foes. Especially MMOGs make it possible to continue playing with the same people in the persistent world.

Because there are several players in multiplayer games, they require some way to coordinate the game and keep the players connected. If there is no control in a game, it can turn into a complete mayhem where players do not agree on how the world is coordinated. One way to coordinate is to use a server. Another is to use a peer-to-peer system. Most games use a client-server model nowadays. Therefore, we wish to examine how well a peer-to-peer system handles game coordination.

Research in the peer-to-peer architecture has focused on MMOGs. One reason why peer-to-peer is interesting for large scale games is the removal of a single server, that potentially can end up being a bottleneck. Thus, a peer-to-peer solution means the maximum amount of players is not limited by the capacity of the servers hosting the game.

When players have full knowledge of events in a game world the game is said to use global consistency. Games that use this approach do not support many players as the amount of updates rise significantly as the amount of players increase. To cope with large numbers of players, players only receive updates about a subset of the game world. This approach builds on the assumption that players are only interested in events in their vicinity. Games using this approach are said to use local consistency.

Knutsson [1] and *Donnybrook* [2] each offers a scalable solution using a peer-to-peer architecture. Knutsson focuses on large MMOGs where the num-

ber of players is important. This is achieved by partitioning the game world and forming interest groups for each partition, thus utilizing local consistency. Donnybrook is a system designed for fast-paced games where global consistency is required, but the emphasis is also on scaling. However, the technique employed causes players to receive full updates of only a small subset of participants. The remaining players send updates once each second.

In commercial games, the peer-to-peer architecture is found in some games. However, it is often a hybrid between peer-to-peer and client-server. The developers supply servers for matchmaking. When a group of peers is matched, one is designated as host. This peer is a client but also acts as the server, responsible for controlling the game state. The host is said to be a super-peer. The difference between a client-server architecture and the super-peer is that the super-peer can change in a game session while the server of the client-server architecture is always the server. *Warcraft III* [3] custom games uses the super-peer architecture.
Demigod [4] is an example of a game where the developers initially implemented a pure peer-to-peer architecture. Before a game starts each peer connects to everyone else in a lobby. Brad Wardell mentions in [5] some of the initial problems related to the pure peer-to-peer architecture. E.g. players unable to make and keep connections to each other through a NAT.

1.1 Problem Definition

The predominant network architecture for multiplayer games is the client-server architecture. In this thesis we examine whether or not it is feasible to employ a pure peer-to-peer architecture.
We wish to examine how well peer-to-peer works in a fast-paced action game with up to 10 players while keeping the game consistent. The number 10 is chosen because similar games like DotA [6], Bloodline Champions [7] and Heroes of Newerth [8] all have a limit on concurrent players in a game of 10.
The game should be playable with an average internet connection. According to Glenn Fiedler 99% of households with an internet connection in the USA, Europe, Japan and Korea have 256kbps download or greater and 93.9% have 256kbps upload or greater as of March 2010 [9].
One of the challenges in a peer-to-peer architecture is how to keep the game state consistent across peers. A game in which a player cannot interact with the other players due to consistency problems is not enjoyable. We wish to limit the delay imposed on the game, such that it is low enough for the average person to not notice the game is being delayed. Another interesting problem is how to handle variables that require mutual exclusion. The solution must not ruin the fast-paced nature of the game.
To sum up, this thesis addresses the following problems:

- Is it possible to keep a fast-paced game with up to 10 players consistent using a peer-to-peer network architecture?
- Can we limit the bandwidth usage such that the game is playable in a normal household? – 256/256kbps.

- Is it possible to make the game work with network latency without players noticing the delay?
- How can we make updates to shared mutual exclusive objects in a fast-paced peer-to-peer action game?

We developed the test game Rawrlocks, a game designed such that we can examine the aforementioned problems. Initial analysis along with a prototype of Rawrlocks was produced during the DAT5-project period resulting in the report "Peer-to-Peer Middleware for Fast-Paced Computer Games" [10]. The result was that Rawrlocks had the desired game mechanics implemented. Two peers could connect and play against each other. Although, there was no guarantee that the game state was consistent. We decided to split up the distribution of game state objects in two modules: A module coupled with the game client used to distribute variables that required fast delivery and an independent module, the synchronisation service, that was used to distribute variables that required reliable delivery. The synchronisation service had basic functionality for object synchronisation.
During this semester, we further develop the modules of Rawrlocks to make it more suitable for its purpose. Testing of Rawrlocks helps provide answers to the presented problems.

1.2 Report Structure

Chapter 2 analyses various parts of how a game works in relation to what is required to distribute a game. The chapter also analyses how latency affects games and outlines other problems found in a distributed game.
Chapter 3 describes the parts needed to make a peer-to-peer multiplayer game work. Starting with how peers discover each other to how game clients communicate. The chapter also explains how latency can be handled in a peer-to-peer game and provides possible solutions to latency hiding.
Chapter 4 explains low-level details of the Rawrlocks implementation. Included is the packet structure of the game client packets and the synchronisation service packets.
Chapter 5 presents the test environment along with the test results. The results are analysed and discussed in this chapter.
Chapter 6 concludes the work and results. It also presents discussion on how the project relates to other games and some ideas for future work.

2 Analysis

This chapter presents analysis of the topics relevant for the problems defined in the Problem Definition 1.1. First, Section 2.1 and Section 2.2 explain the concept of "game state" and what consistency means for game state. Section 2.3 describes three different connectivity models. The models determine how objects are distributed in a multiplayer game. Multiplayer games require that messages are sent back and forth. Thus, the messages are affected by network latency. Section 2.4 contains a description of a method on how to estimate one-way latency.
Next, Section 2.5 describes the game mechanics in Rawrlocks. Some of the real world problems related to this topic, which are not adressed in this thesis, are briefly covered in Section 2.6. Lastly, Section 2.7 presents four techniques that are used to achieve consistency in the Source engine.

2.1 Game State

To make a game progress, it must be possible to change the state of the game. The state of a game consists of the state of the set of objects in the game. In order to define what the game state is, we define what an object is.

Definition 2.1

Object: A collection of related variables

There are two types of objects – mutable and immutable objects. The state of the game depends on the mutable objects. If Chess is used as an example, the mutable objects are the 32 pieces and who's turn it is. The variables of the mutable objects can be modified during a game.

Definition 2.2

Mutable Object: An object that can be modified

The only immutable object in Chess is the board, and it is therefore not a part of the game state. The board does not change state during a game, but contains information the pieces use to verify whether or not a move is inside or outside the board. It is instantiated at the beginning of the game.

Definition 2.3

Immutable Object: An object that cannot be modified

In order to have a complete game state in a game of Chess, a peer must know the state of every object. As mentioned each Chess piece has two game state variables: Its position and whether or not it lives. The state of the current turn is either black or white. A new game state occurs whenever a player makes a move because the state of at least one of the 32 objects changes. Thus the state of the game differentiates from what it was. Two players is able to continue a game of Chess if and only if they have the same complete game state.

Definition 2.4

Game State: The current state of each mutable object in a game

Updates to mutable objects must be verified and distributed by a single point. Otherwise peers can update objects at will, and disagreements of the object's state is likely to occur.

Definition 2.5

Object ownership: Any mutable object is owned by exactly one peer. Ownership implies that only the peer that owns the object is authorised to update the state of the object

Definition 2.6

Replicated Object: A copy of an object which the player does not control

The players who does not control a given object, have a replica of the object.

Chess is an example of a turn based game. The rate of updates to the game state depends on how much time each player spends per move. In a real time game, however, the rate of updates are at a constant pace, typically several times per second.

2.2 Consistent Game State

It is important to maintain a consistent game state. Therefore, peers must agree on the state of each mutable object. Disagreeing means that conflicts can arise and result in a player's action being legal according to the local game state but illegal at another player.

Definition 2.7

Consistency: A game is consistent when replicated data ends up equal across peers when modified.

The game state does not have to be equal at the same absolute time across participating peers.

Definition 2.8

Absolute time: The time according to a global clock

A consistent gameplay is a gameplay where all events occurring in the game state of one peer, occurs across every peer.

2.2.1 Consistency Affected Properties

There are three properties that affect the consistency of a variable; freshness, exactness and ordering. These properties cannot all be handled at the same time. It is therefore important to evaluate which properties are most important for a variable. Before evaluating the properties on the game object variables, the properties are defined:

Freshness How new and up-to-date a variable must be. A high freshness requirement means the variable must be sent and handled as fast as possible. A low freshness requirement means that the variable does not need to be updated frequently.

Exactness How exact a variable must be to be usable. A high exactness requirement means that the variable must be consistent across peers and the variable is not allowed to reach a sustained conflicting state. A low exactness requirement means that the variable is allowed to have variations across peers.

Ordering How important sequential execution of variable updates is. A very high ordering means that execution of updates must be linearisable. A linearisable execution order means that events are executed in the exact order they are generated as if it is a synchronous system. Variables that require lesser ordering can rely on sequential ordering. Sequential ordering means that only updates from the local client must be executed in the order they are generated as if the local system is a synchronous system.[11, P.616] The importance is related to who is able to modify a given variable. If only one peer attempts to change a variable, the result is the same with linearisable ordering and sequential ordering.

Ordering is a special case because it is important to have any game state modification seem linearised to have a consistent gameplay across clients. If a game is not linearised, clients might see the same situations played out differently. The difference between true linearisable execution and making the execution seem linearisable is how strict the constraint is. It can be expensive timewise to enforce linearisability on every action because it must be ensured that there are no other actions preceding the next action in the execution chain. The game requires real-time execution. Trying to execute in a linearisable order and ignoring any errors is a cheap solution because execution happens in the current known global order. Errors can be ignored where there are ownership constraints on variables. With such constraints, updates only occur at one peer and therefore the game state eventually becomes consistent again.

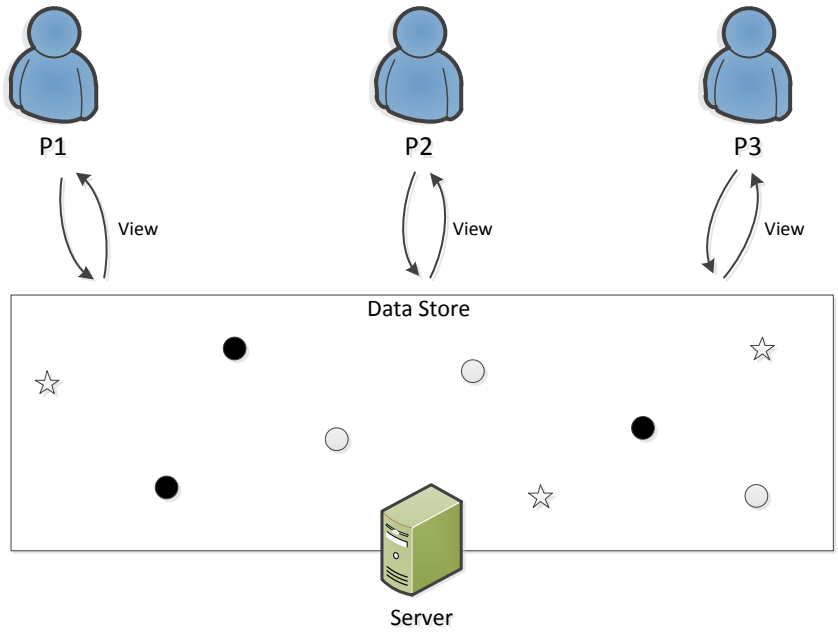


Figure 2.1: The network topology of the client-server network architecture. The peers are connected to a server that controls the whole data store with the game state.

2.3 Connectivity Models and Object Distribution

There are three solutions for managing the mutable objects in a multiplayer game: Client-server, peer-to-peer and super-peer. This section explains the idea behind each solution.

Client-Server

In a client-server solution all clients connect to the same central server. The situation is depicted in Figure 2.1. The server has the capacity to host game sessions[1]. The game state resides in the data store on the server. The server acts as a master that holds the true game state and has supreme ruling over any changes to mutable objects.

Peer-to-Peer

In peer-to-peer solutions all clients connect directly to each other, see Figure 2.2. It is possible to use several network topologies, e.g. fully connected or ring topology. However, for this discussion we assume that a fully connected topology is used. Each peer is responsible for managing a subset of the mutable objects.

[1] A game session is a single game from beginning to completion e.g. one match of chess.

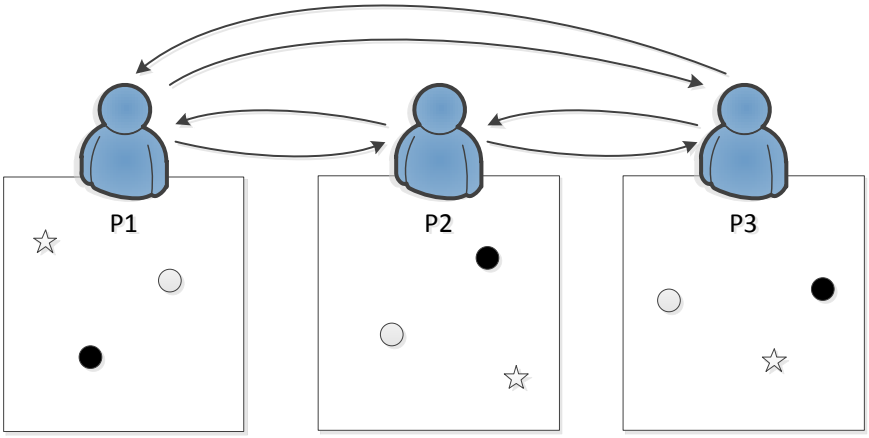


Figure 2.2: The network topology of the peer-to-peer network architecture (fully-connected). Each peer have a local data store where they control part of the game state.

For instance, in a game of chess the mutable objects are the 32 pieces and the question of who's turn it is. Each object must be owned by one peer. The mutable objects are evenly distributed to the peers, such that each peer has a set of chess-pieces and one peer has object ownership of who's turn it is.
There is a risk that an object owner disconnects and leaves an object ownerless. In order to avoid this situation, objects can be replicated and stored on other peers. If a client who owns an object disconnects, a peer with an up-to-date replica of the object in question, can assume ownership of the object. The peer uses the last stored values of the object and restores the game to a consistent state.

Super-Peer

A super-peer solution resembles a client-server solution. The server is a single peer – super-peer. The super-peer can change during the game session if, for instance, the current super-peer disconnects or has network problems. In some games the super-peer is the person who created the game session. This solution is cheap from the perspective of the game developer as the users of the game handles the hosting.

2.4 Latency in Games

One of the biggest problems in games with multiple participants playing online is the network latency between the participants. According to Sears and Jacko [12] network latency is affected by bandwidth, distance, routing hops and jitter.
Bandwidth is the amount of data that can be transferred to and from a computer over time (upload and download). *Smed & Kaukoranta* [13] defines la-

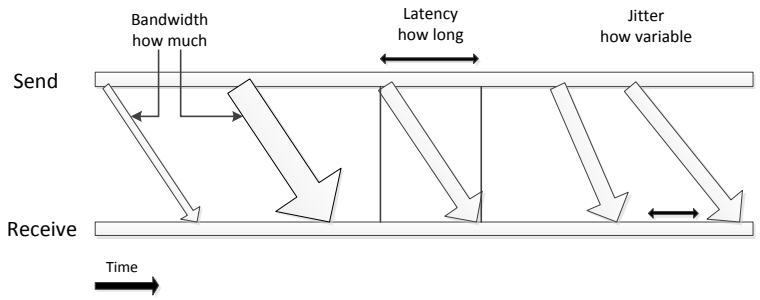


Figure 2.3: Difference between bandwidth, latency and jitter [12, P.336]

tency as the amount of time it takes a packet to be sent from the originating node to a destination node – this is determined by distance and internet infrastructure. Jitter describes the variance in latency over time. Figure 2.3 illustrates the relationship between bandwidth, latency and jitter.

Definition 2.9

Network Latency: The total time it takes a packet to be transmitted from one computer to another

Determining one-way latency precisely is a difficult task, but it is commonly estimated to half the round trip time (RTT). As network latency grows it becomes difficult to maintain a game that appears consistent for the participants. In our 9^{th} semester report [10] we established that humans do not notice the delay between an action occurs until he sees it, if it is small and hidden enough. We found a 100ms limit in the previous work [10, Sec. 2.4.3] and can also be found in [14]. We choose to make the game work with a latency of 50ms because that limit covers Europe, as described in [10, Sec. 2.3.2] and is below the maximum of 100ms.

Several methods to hide latency exist. One of the reasons that latency hiding techniques work is that a player cannot see the rendered game of the other participants. When a player cannot see the other monitors directly, different game states can only be noticeable through conflicting actions in the game. E.g. an action would be legal according to the game state of one player, but conflict according to another player's game state. Therefore, any change considered legal by the game logic does not reveal any latency.

Latency is important to handle correctly. One reason is that the game can reach an inconsistent state if latency is not handled correctly. It can also give an unfair advantage to some peers. An unfair advantage could be knowledge about changes to the game state before another player has knowledge about the updates. E.g. a power-up respawning – if a player knows about a power-up respawning before other players, the given player has that much more time to try and obtain it. A more interesting example is the position of an avatar and the possibility to evade fireballs from other players.

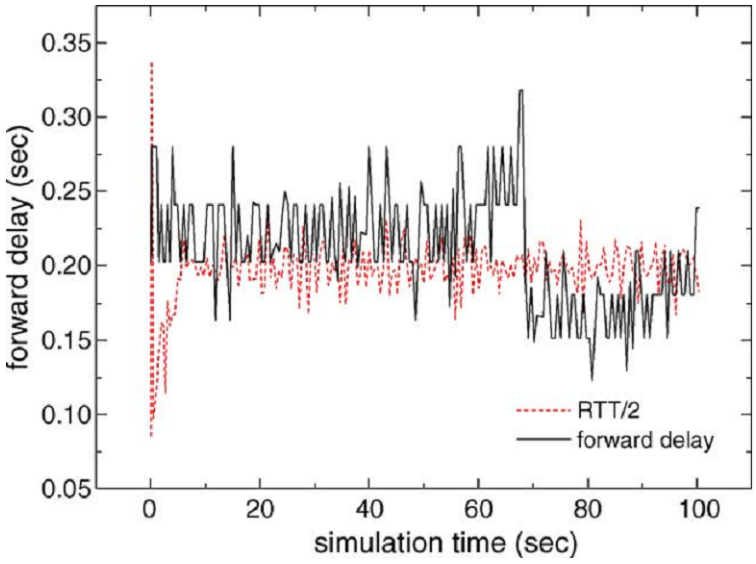


Figure 2.4: Figure taken from *One-way delay estimation and its application*[15], shows estimating latency with RTT/2 versus their solution.

2.4.1 One-Way Latency Estimation

Finding the exact one-way latency is difficult when there is no global clock. Using RTT/2 is inaccurate as the latency is not necessarily the same both ways. Several obstacles can cause the latency to differ forward and reverse. E.g. different routes through the Internet each way or throttling by a cable modem due to asymmetric connection.[15].

One possibility is to measure the latency more exact by marking the data packet and then return an ACK packet. This only works for TCP and does not give the exact latency, but yields a better result than using RTT/2. A more exact measurement of latency gives a better end result, but even current TCP implementations use RTT for congestion control.

Using the more exact latency measurement would further show which direction is causing the slowdowns. As seen in Figure 2.4, RTT/2 does not deviate much from the forward delay. For the most part the deviation is up to 25% with a worst case of 50%. In Rawrlocks context, with a maximum latency of 50ms, this would yield a worst case deviation of 25ms and normal up to 12.5ms. This is an error margin that cannot be removed. The deviation is not taken into consideration as it is impossible to determine with the available information.

2.5 Game Description

Rawrlocks is a fast-paced arena-based action game. The game is loosely based on Warlock Brawl [16] and Bloodline Champions [7]. Rawrlocks is a round based game with 2-10 players starting in a single shared 2D arena where the

players fight each other. An illustration of the arena is shown in Figure 2.5. When only one player is alive, that player is declared the winner and the round ends. The arena consists of an area with walkable ground and an area with lava that damages any player standing on it. At the beginning of each round the arena has a fixed size that decreases as time passes, such that, in the end, the entire arena is gone. Each player controls a single avatar that can use abilities and moves around in the 2D world using the keyboard and mouse. The arena contains two power-ups that enhances certain features of the avatar for a period of time.

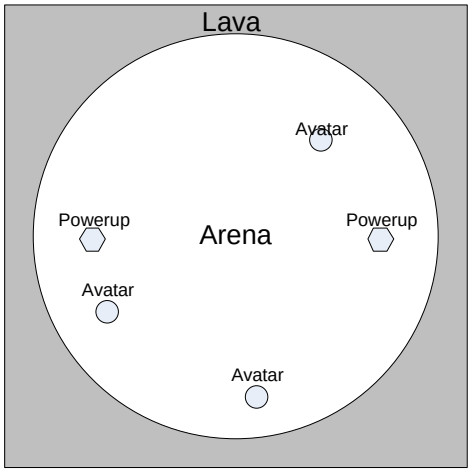


Figure 2.5: The arena with three players and two power-ups

Avatar

An avatar has a predefined amount of health. Health is decreased if the avatar is hit by offensive enemy abilities or if the avatar touches the lava. When an avatars' total health becomes equal to or less than zero, the avatar dies and loses the round.
An avatar is able to fire abilities, get hit by offensive abilities, pick up power-ups and move around on the arena. Abilities have a cast time. The offensive abilities affect an avatar with a knockback.

Cast time Cast time is the time interval between a player activates an ability to the time the ability is fired in the game world. The value of cast time varies from ability to ability, but is constant per ability.

Knockback The knockback forces an avatar to move along a vector – the *knockback vector*.It is the movement of an avatar that is manipulated, and therefore the effect can be partly negated by moving in the opposite direction. The knockback effect decreases over time and is gone after a short time period.

Abilities

An ability is an action the avatar can perform. The abilities are divided into offensive and defensive: An offensive ability deals damage and applies knockback to opponents. The offensive abilities are fireball and scourge. A defensive ability manipulates an avatar's state and helps the avatar to defend himself in a number of ways. The defensive abilities are teleport and thrust. The four abilities are as follows:

Fireball The avatar sends a fireball in a direction indicated by the cursor's position. The fireball moves in a straight line starting from the avatar with a constant velocity for a limited range. If an enemy avatar is hit by a fireball, the avatar loses a fixed amount of hit points and has knockback applied in the direction of the fireball.

Scourge The avatar creates an explosion at its current position in a circular area around the avatar. The explosion deals damage and applies knockback to enemies within range of the ability. The knockback is applied in a direction away from the avatar that activate the scourge ability.

Teleport Changes the avatar's position to the position of the cursor projected down onto the arena. The teleport is limited to a certain range.

Thrust Applies a knockback to the player's avatar in the direction of the cursor. The force of the knockback is constant no matter the distance between the cursor and the player.

Power-Ups

Rawrlocks has two power-ups. These power-ups can be picked up by avatars and disappears from the arena once picked up. Hence they can only be picked up once per spawn. The effect given to the avatar is temporary and is either doubles the movement speed of the avatar or doubles the damage of the avatar's offensive abilities. After a fixed amount of time the power-up respawns at its initial position.

2.6 Other Real World Problems

In a real world environment there are several problems affecting a multiplayer game. These problems have not been taken into consideration in this thesis. The problems must be solved to make peer-to-peer games usable in a real world scenario. Therefore, this section briefly explains the problems that are relevant.

2.6.1 Network Address Translation

NAT (Network Address Translation) is currently widely deployed. More than 90% [17] of all broadband subscribers deploy NAT on their home network. In Rawrlocks, all peers connect to each other to create the correct peer-to-peer

network for gaming. Therefore, a way through the NAT must be found as to connect the NAT'd peers. The easiest way to a NAT'd computer is for the user to manually forward a port or use UPnP (Universal Plug and Play) to forward a port on request. UPnP is not supported by all routers.

The peers that do no forward ports manually or through UPnP must deploy other methods. One method to get through a NAT is NAT Punching, but this method only has a 90% success rate [18]. This means that other methods are required to connect the last 10% of the peers. Another suggested method is to use super-peers which proxies the traffic among the peers unable to connect with each other. This adds one extra hop to any packets sent via the proxy-peers, but ensures everyone is able to connect in a peer-to-peer network as required by Rawrlocks. A hop is the delay it takes to send a packet from one peer to another over the network.

2.6.2 Peer Disconnects

In a real world scenario peers can leave the game unexpectedly. The result is loss of variables and a stale avatar in the game. A game must be able to continue even with peers disconnecting, otherwise the game experience can be ruined by a single peer. Before a peer can be removed from the game, the peer must first be marked as not participating in the game anymore. Packets are sent constantly in games. If a peer suddenly stops sending game updates, it is expected that the peer is not part of the game anymore. The participating peers can inform each other when a peer stops sending game updates. If more than half observe this situation, the peer can be disconnected. The method is basically an election where a majority of peers may agree to vote a peer out of the game.

When a peer is disconnected and must be removed from the game. There are two things that must be done.

- The avatar must be removed from the game.
- Ownership of the variables controlled by the peer must be re-delegated to other peers.

Removing the avatar is fairly simple as it is just deleting the avatar object on all the peers. This must be done when the disconnect has been decided.

In a game like Rawrlocks where everyone are against everyone, it does not pose as a big competitive advantage or disadvantage that one player disconnects.

If a peer disconnects after a very short time with no data, small hickups in the network can easily ruin the game for an unlucky player. Warcraft 3 solves this problem by not disconnecting the peer after a few seconds with no data. It freezes the game and shows all the peers that the game is waiting for the last peer. The upside to this solution is that it is possible to wait longer without ruining the game. The downside is that all the peers must wait for a single peer.

2.6.3 Peer Connectivity Issues

A common connectivity problem is a routing problem. What happens if peer 1 and peer 2 are unable to communicate with each other, but peer 3 to 5 have no problems communicating with everyone? Peer 1 and peer 2 causes the game to become inconsistent when they update their local variables. E.g. location and variables owned by them. The solutions to this problem are not cheap and routing problems can easily result in the game splitting into several "sub-games". The simple solution is to route traffic from peer 1 to peer 2 through another peer, creating a super-peer type scenario. The number of hops becomes two instead of one and part of the advantage in making the game peer-to-peer is lost.

The important thing to note is that connectivity issues are more likely to push over a game like Rawrlocks when a peer-to-peer model is used compared to a server-client model. This is because the number of connections is larger than with a server-client model and therefore has more points of failures connectivity-wise.

2.6.4 Cheaters

For all products there are someone who will try to break it and use it in ways not originally intended. In multiplayer games it is often used to gain benefits over other players, also known as cheating. Since peers control part of the game state it opens up for a whole new category of cheats compared to the client-server model.

Anti-cheat solutions directly oriented towards peer-to-peer games have been analysed in the works of *Baughman and GauthierDickey* [19][20]. The focus is on correct ordering of the correct events and through that prevent cheat by predicting the future. A correct event is an event not tampered with after it has taken place. Rawrlocks does not ensure each event reaches all peers, thereby making it hard for a scheme like this to work in practice. The best way to ensure no cheat is to use methods that client-server models employ. This could be anti-cheat software on the peer and detecting abnormalities too big to be possible in the game.

2.7 Multiplayer Networking Techniques

This section examines the Source engine and how it makes networked games seem consistent and real time to the players. The section is based on [21]. Source employs a client-server networking architecture, but some of the techniques can be used in peer-to-peer solutions.
The Source engine uses four methods to cope with the issues introduced by network latency. All these methods are invisible to the player:

- Data compression
- Interpolation
- Prediction
- Lag compensation

2.7.1 Data Compression

In order to reduce the required amount of data that must be sent, Source uses *delta compression*. Therefore, the server does not send a full snapshot of the game state, but only changes since last acknowledged update. The server only sends a full snapshot at the beginning of the game or when a client suffers from heavy packet loss for a couple of seconds.

2.7.2 Interpolation

The Source engine sends a fixed amount of updates per second to clients by default. If the objects were only updated when new information is received and set exactly to received, moving objects and animations would look choppy and jittery. Therefore, the position of an object is interpolated between snapshots. If x_t is the position at time t, and if $x_0 = 0$ and $x_1 = 1$, the interpolation sets $x_{0.5} = 0.5$. This means that at times where the server does not inform a client where an object is, the client calculates the position of the object with the help of already received values.

2.7.3 Prediction

Source lets clients predict their actions locally. The effect is that the local client has a predicted state of the local game state while the server has the true state. If the two states differ, the client has made a prediction error that must be corrected since the server has the true state. As the latency from client to server increases the predictions are more likely to be incorrect. If the value is being corrected by the server, the client interpolates from the locally predicted position to the real position received from the server. At high latencies this behaviour is likely to produce erratic behaviour.

2.7.4 Lag Compensation

In a Source game if a player activates a key, the server receives the message after latency time. If the player aims at an enemy on his screen and fires a shot, the enemy may have moved on the server before the message is received. The result is that the local player believes he hits the enemy, but this is not true according to the server's game state. This is because the enemy has moved away from the shot in the time it takes for the message reach the server. In order to fix this issue, the server keeps a history of all recent player positions for one second. If a user command is executed, the server estimates at what time the command was created, t_e:

$$t_e = \text{Current Server Time} - \text{Packet RTT} - \text{Client View Interpolation}$$

The server moves all other players back to where they were at the command execution time. Therefore, the position of the enemy is the same on the server, as it was when the player fired his shot towards the enemy on the player's screen.

3
Design

This chapter describes the design of Rawrlocks. Section 3.1 covers a method to discover other peers. Next, the general architecture of Rawrlocks is outlined in Section 3.2, and the components of particular interest are explained more thoroughly in Section 3.3 and Section 3.4. Lastly, Section 3.5 and Section 3.6 presents several design suggestions to cope with the problems caused by latency. In the end the aspects required to develop a peer-to-peer multiplayer game has been covered.

3.1 Lobby Server

Typically, multiplayer applications require a rendezvous point where peers can discover each other. File-sharing systems such as *BitTorrent* [22] require tracker servers to facilitate discovery of other peers. We mentioned in the introduction to Chapter 1 that some game developers supply servers for matchmaking, but game sessions use peer-to-peer or super-peer.

Thus a rendezvous point for the peers is required – a lobby server. A lobby server needs only basic functionalities, such that it is easy and fast to start a game session. For instance, peers must be able to create rooms, such that peers in one room play with each other. Peers must get the appropiate information required to establish connections and create the peer-to-peer network. The functionalities of the lobby server are as follows:

Handle connections A peer must be able to connect to the server and be identified by a unique ID, e.g. their name.

Room Rooms are used as a lobby for the peers that takes part in the same game session and provides a segregation of peers. A room contains a limited amount of peers. Any peer is able to create and join rooms. The rooms are hosted on the lobby server.

List of Room The server keeps track of all rooms. Rooms are removed once the game has started. The list of rooms is retrievable by peers.

Start game When a game session starts the server sends each connected peer a client list with ID, name and IP address of the peers in the room. This

information is used by Rawrlocks to establish connections between the peers.

The lobby server uses a client-server network architecture and is shown in Figure 3.1.

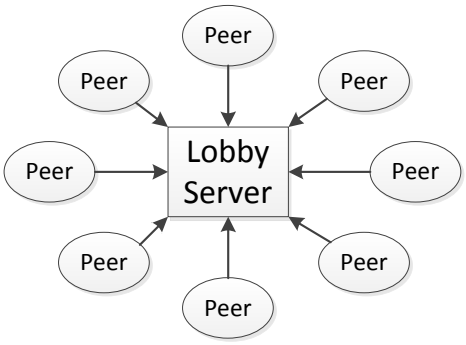


Figure 3.1: The network architecture of the lobby server.

3.2 Rawrlocks Architecture

This section explains the architecture of Rawrlocks. Figure 3.2 depicts the overall architecture. The architecture is divided into several layers, some of which are split into modules.

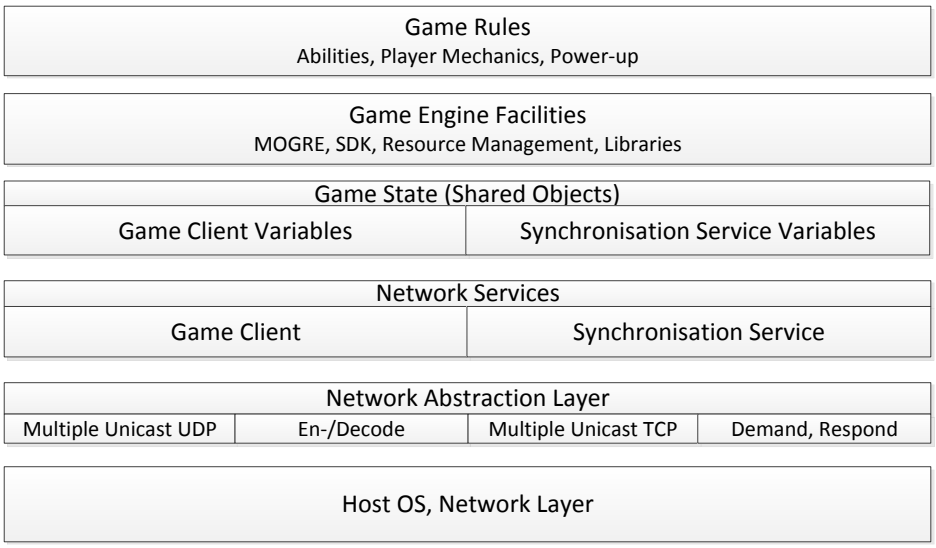


Figure 3.2: Overall architecture of Rawrlocks.

3.2.1 Game Rules

This layer represents the game specific logic in Rawrlocks. All game specific events are handled in this layer. The logic behind the game decides how fast avatars move, how fast projectiles fly, how far an avatar is knocked back. The limited physics that is implemented in Rawrlocks is also in this layer. E.g. the game logic observes an avatar colliding with a power-up. After this event the colliding avatar has the power-up effect applied onto him and the power-up itself is removed from the arena. Everything that makes Rawrlocks a game is handled in this layer.

3.2.2 Game Engine Facilities

A lot of software programs make use of libraries and SDKs – computer games are no different. This layer represents the game engine and SDKs used to power Rawrlocks. For Rawrlocks we use Mogre, which is a .NET wrapper for Ogre3D [23], and MOIS as input system. Mogre handles tasks such as rendering, resource- and scene management. Since this layer mainly contains the utilities for the execution of the program, it is not of much interest with relation to the scope of this project.

3.2.3 Game State (Mutable Objects)

This layer represents the game state in Rawrlocks. The game state in Rawrlocks is the combined state of every object in the game. The only immutable object in Rawrlocks is the lava floor and is not discussed further. When the game is designed, the implementor must classify the objects according to freshness and exactness and decide how to distribute the objects of the game. The objects are divided into two categories that determine which module handles the distribution:

- Objects distributed using the game client (GC) – these objects have high requirements on freshness.
- Objects distributed using the synchronisation service (SS) – these objects have high requirements on exactness.

Game state objects are further discussed in Section 3.3.

3.2.4 Network Services

The segregation of the distributed variables in Rawrlocks require two different network services for distribution. One network service to ensure fast delivery – this service is an integrated part of the GC. The other network service must provide reliable delivery. This service is designed as an add-in such that it can be used by games similar to Rawrlocks. The network services are further discussed in Section 3.4.

3.2.5 Network Abstraction Layer

As mentioned the two network services handle distribution of game state variables in Rawrlocks. Each network service needs a transport protocol for this end-to-end communication. Hence, the network abstraction layer represents the encoding, decoding and transport of data.

3.3 Game State Objects

Game state objects are constantly changing in Rawrlocks. This section outlines the game state objects and their variables and classifies the freshness and exactness requirements of each of the variables.

In Rawrlocks changes to a variable symbolises that an event has happened. Events are divided into two groups, that determines the cause of when variables are updated:

Definition 3.1

Triggered Events: Events activated according to game rules.

Definition 3.2

Activated Events: Events activated by player input.

The variables of each object in Rawrlocks are described and categorised below. Furthermore, we classify whether the alteration of a variable is bound to a triggered or an activated event.

Power-Up

The variables bound to a power-up object are:

Availability Describes whether the power-up is available to be picked up. This variable is altered by a triggered event, i.e. an avatar colliding with the power-up

Active Is it active on an avatar and, if it is, which avatar is it active on. This variable is altered by a triggered event, i.e. has the power-up effect been active for its designated duration.

Both variables on a power-up has high exactness requirements and low freshness requirements. Therefore, they are distributed via the SS. Changing one variable of a power-up object means changing the other. Thus, the variables of a power-up must have the same owner.

Restart Round

The variable determines if a round should be restarted or not. Restart round is a triggered event. The event is triggered if less than two players are alive. The peer who owns the variable decides whether or not the condition is fulfilled.

However, it is desirable that all peers see that the condition is fulfilled. It is necessary that the event restart round is exact across peers, since this event resets arena size and variables on the avatar object. Therefore, we create a SS variable in order to keep track of whether a new round must be started.

Arena

The variable bound to the arena object is:

Size The size of the arena must be exact. If an update to arena size is lost, an avatar can stand in lava at one peer while not standing in lava locally because an update was lost. The arena size is reduced by a triggered event that is triggered at fixed time intervals. The size of the arena is distributed using the SS.

Avatar

The variables bound to an avatar object are:

Avatar health Altering the health of an avatar is a triggered event that occurs whenever an avatar is hit by an enemy ability or is standing in the lava. The health of the avatars must be exact, while the freshness is of minor importance. For these reasons health is distributed using the SS.

Last damage dealer The last damage dealer, and in effect, who gets the kill if an avatar dies is a value that must be exact. This is a triggered event that occurs whenever the health of an avatar is reduced by an enemy avatar. This event is heavily related to avatar health since it is used to decide avatar kills. It is therefore distributed using the SS.

Avatar specifics This covers, among others, the position, movement vector and mouse position. Each of these values must be known in order to position the avatar correctly across peers. The avatar specifics has loose exactness requirements while freshness is very important. Each of the variables distributed in avatar specifics are activated events. Due to the high freshness requirements the avatar specifics are distributed using the GC. The peer controlling the avatar propagates the message to the other peers in the game.

Ability cast When an avatar is in the process of activating an ability, it starts casting it. This events sets two variables on the avatar: Whether the avatar is casting, and, if it is, which ability it is casting. Players use this knowledge to predict other players' moves. Ability cast is an activated event with high freshness requirements. It is therefore distributed using the GC.

Ability fire When an ability is fired the opponents must know about it to be able to react to the incoming ability as fast as possible. The activation of an ability requires exactness in the sense that the other players in the game must see the ability being activated. When an avatar has casted an

Variable Distribution					
num.		Fresh	Exact	Event type	Placement
1	Power-up availability	L	H	triggered	SS
2	Power-up active	L	H	triggered	SS
3	Restart round	L	H	triggered	SS
4	Arena size	L	H	triggered	SS
5	Avatar health	L	H	triggered	SS
6	Last damage dealer	L	H	triggered	SS
7	Avatar specifics	H	L	activated	GC
8	Ability cast	H	L	activated	GC
9	Ability fire	H	H	triggered	GC
10	Ability hit	H	H	triggered	GC

Table 3.1: The game state variables in Rawrlocks and their freshness and exactness requirements (L is low, H is high), whether the variable triggers an event and if it is distributed using the SS or the GC.

ability for its cast duration, a triggered event fires the ability. It is worth noting that this event does not alter any game state variables with high exactness requirements, so losing the event does not create an inconsistent state. This event is distributed by the GC by the player who fired the ability.

Ability hit While the health of an avatar has loose freshness requirements, the fact that they got hit has high freshness requirements. This events does not change health, because the health reduction already happened before this event is activated. This event is the notice to other peers that an avatar was hit by an ability. Like the firing of an ability, this value must be exact. If an ability hits an opponent this event is triggered. This event does not alter any game state variables with high exactness requirements, so losing the event does not create an inconsistent state. Because of the high freshness requirements, this event is distributed by the GC by the player who fired the ability that hit.

3.3.1 Variable Classification

Table 3.1 shows an overview of all the game state variables, the level of their freshness and exactness requirements, whether or not it is a triggered event and whether it is distributed using the SS or the GC.

Rawrlocks has a total of ten variables that are shared among peers. Figure 3.3 shows the freshness and exactness requirements of each variable. The X-axis determines how important the freshness requirement of a variable is, going from left to right with higher to lower importance respectively. The Y-axis determines how important the exactness requirement of a variable is, going from bottom to top with higher to lower importance respectively.

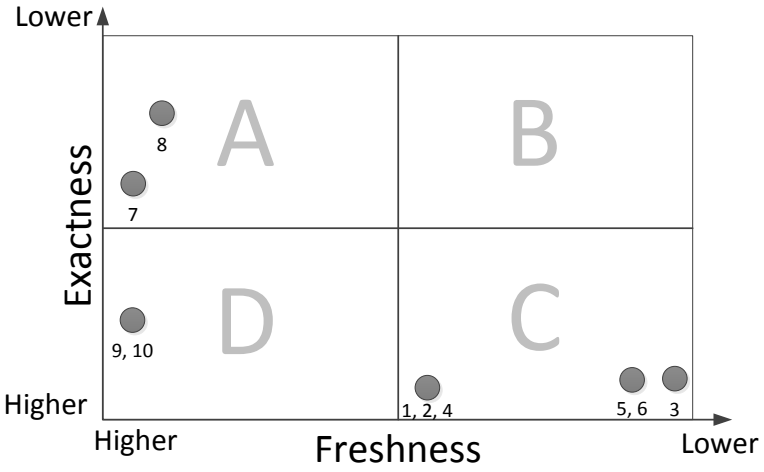


Figure 3.3: Exactness and freshness of Rawrlocks variables.

Each circle represents one to three variables depending on the numbers below. The numbers in turn relate to the number of the variables in Table 3.1. E.g. number 1 represents Power-up availability, which is located close to the origin of the Y-axis and approximately halfway out on the X-axis. Thus, exactness is of high importance while the freshness requirement is lower.
The figure is divided into four areas: A, B, C and D. Each area helps determine which module should handle a given variable. Area A contains variables where freshness is of highest importance. These variables are distributed by the GC. Area B is a grey area, and variables within this area could be distributed by either module. Rawrlocks however does not contain variables in this area. The variables in area C has high exactness requirements and must be distributed by the SS.
Area D contains variables, that must be both fresh and exact. In Rawrlocks, two variables have high exactness and freshness requirements – ability fire and ability hit. Both variables have higher requirements for freshness than for exactness. Therefore, they are distributed by the GC. Because of the fact that neither ability-fire or ability-hit alters any variables with high exactness requirements, these events require a loose form of reliability. If one packet is lost, the event should not be lost. However, if a peer loses packets until a new event of the same type is activated, the event is lost.

As mentioned the variables are distributed using either the GC or the SS. These two modules are located in the layer below the game state layer, and are described in Section 3.4.

3.4 Network Services

This section describes the design of the two networking services in Rawrlocks. Section 3.4.1 describes the design of the GC while Section 3.4.2 describes the design of the SS.

3.4.1 Game Client

The GC's task is to distribute the variables that were defined to be handled by the GC in Table 3.1. These are the variables that have high requirements on freshness.

Fast packet delivery is achieved by using the UDP protocol. The recipient must be able to update a local replica that simulates the behaviour of the player sending the packet no matter how many packets are lost in the meantime. Furthermore, due to the high freshness requirements on every GC variable, the values must be updated and sent at a high rate.
In order to meet the high exactness requirements on the triggered events of the GC, the activation of an these events triggers instantly sending a packet to every peer in the game.

3.4.2 Synchronisation Service

The SS is the module that keeps high exactness variables consistent across peers. The SS is an independent component and is not directly integrated into the game but runs simultaneously. The SS has two communication types – one with the local GC and one with the other SSs in the game. The SS is in constant communication with the game to make sure updates arrive as fast as possible.

The SS works as a variable storage where the GC can request updates to variables. The SS informs the GC of any updates to any variable currently stored.

Variables are restricted in order to simplify the SS and define a predictable behaviour to keep the SS fast and its complexity low. The SS has the following restrictions on variables.

- A variable can only be an integer.
- All variables must be initialised by the game client before the game starts.
- The local GC initialises all variables in the same order across SSs.
- A variable must have exactly one owner.

Allowing only integers removes the need for data type definitions. It is also the only needed data type for the test game. Initialising all variables in the same order at the beginning of the game makes it possible to evenly distribute ownership of the variables without negotiation.

The SS has two commands to change variables, *Set* and *Modify*.

Set (v, x) Changes a variable, v, to x.

Modify (v, x) Changes a variable v, to $v + x$.

The *Modify* command uses delta updates. The use of delta updates lowers the number of hops in a system. If a peer wishes to alter a *Modify* variable,

	Modifiability	Command
Power-up availability	owner	set
Power-up active	owner	set
Restart round	owner	set
Arena size	owner	set
Avatar health	all	modify
Last damage dealer	all	set

Table 3.2: SS game state variables in Rawrlocks and whether or not it is modifiable by owner or all and if the variable can be set or modified.

the peer contacts the owner of the variable and tells him to alter the value of a variable by a specific amount.

The *Set* command sets a variable to a specific value. The use of the *Set* command depends on the ownership mode of the variable. The two ownership modes of variables in the SS is described below.

There are two ownership modes on SS variables:

Modifiable by owner All GCs inform their local SS about changes to SS variables. If a variable is set to be modifiable by owner, only the owner is able to modify or set the variable. This means that variables that are modifiable by owner, are updated according to the game state of the owner of the variable. This ownership type works since peers have full knowledge about the game state and can observe the ingame events that leads up to a change in a variable.
E.g. an avatar is observed by the GC to have collided with a power-up. The variables on the power-up object are grouped together in the SS and are both modifiable by owner. According to game logic, the power-up effect must be applied to the avatar that collided with the power-up and removed from the arena. The GC sends a request to its SS that the aforementioned happened. If the SS that receives this request owns the power-up variable, the request is accepted. If the SS does not own the power-up variable, the request is dropped.

Modifiable by all One GC informs its local SS about changes to a variable in a situation where the GC is designated to observe and inform about a given event. If a variable is set to be modifiable by all, every peer can request changes to the variable but only the owner applies the changes. Therefore, modifiable by all variables never reach an inconsistent state.
E.g. an avatar shoots another avatar with a fireball and the target loses health. The GC request to change a variable and sends this to its SS. If the SS does not own the health point variable of the hit avatar, the SS that owns the health point variable is informed to modify the health point of the avatar that was hit.

The variables in Rawrlocks that are distributed by the SS is shown in Table 3.2. Their ownership mode and the command type that is used to make updates to the variable are defined for each variable.

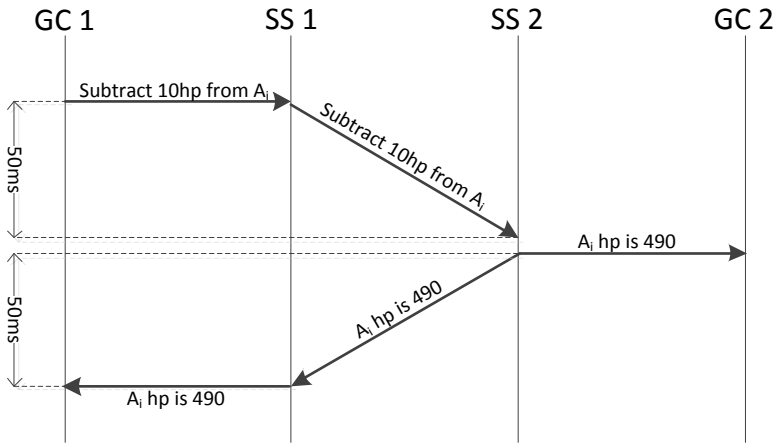


Figure 3.4: Update of variable that uses the modify command and its ownership type is modifiable by all.

Ingame Situations

This section describes three common ingame situations that are used to describe the SS behaviour. The situations use the following definition:

Definition 3.3

A_i**:** Avatar i, the avatar with ID number i.

Situation A: GC1 informs SS1 to decrease the health of A_i by 10 points. This situation is shown in Figure 3.4.

- SS1 checks who owns the health point variable of A_i.
- The owner of the variable is informed about the update.
- The health of A_i is updated according to its ownership mode and command type. The SS owning the altered variable informs every other SS about the updated value.
- SS1 receives the update to the variable and informs GC1.

Situation B: GC1 observes A_i colliding with a power-up. GC1 informs SS1 to remove the power-up and apply the power-up effect on A_i. SS1 is not the owner of the variable. This situation is shown in Figure 3.5.

- SS1 ignores the request since it does not own the health point variable of A_i. The game state remains unchanged.

Situation C: GC1 informs SS1 that A_i is colliding with a power-up. The power-up effect must be applied on A_i and the power-up must be removed from the arena. SS1 is the owner of that particular power-up group. The situation is shown in Figure 3.6.

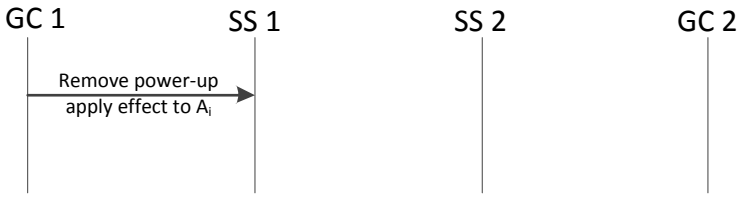


Figure 3.5: Situation where a power-up variable is not owned by SS1.

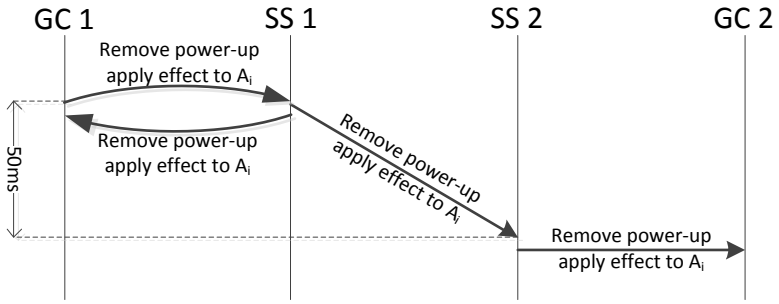


Figure 3.6: Situation where a power-up variable is owned by SS1.

- GC1 informs SS1 about the update and requests the given variables changed.
- SS1 owns the power-up variables and informs all the other SSs about the update. The power-up is removed from the arena, and the effect is applied to A_i.

3.4.3 Network Abstraction Layer

Section 3.4 described the two modules that handle distribution of game state variables in Rawrlocks. Each module needs a transport protocol for the end-to-end communication. Hence, the network abstraction layer represents the encoding, decoding and transport of data.

UDP is already mentioned as providing fast packet delivery for the GC. The GC's main task is to distribute variables with high freshness requirements. Furthermore, UDP's header is only 28 bytes including the IPv4 header [24]. The GC sends packets at a rate of 20 per second. Thus, the GC sends a packet every 50ms. The fully-connected topology implies that each update is broadcasted to all peers. Each second a GC sends: $((n-1)\cdot(20+k))$packets, where n is the amount of players in the game and k is the amount of locally triggered events that occured in the last second.
Figure 3.7 shows connectivity between GCs. Each peer uses a UDP socket to send data to other peers in the game.

The SS requires exactness and assurance that packets arrive. Overhead is less important and a reliable protocol such as TCP is needed. Requests

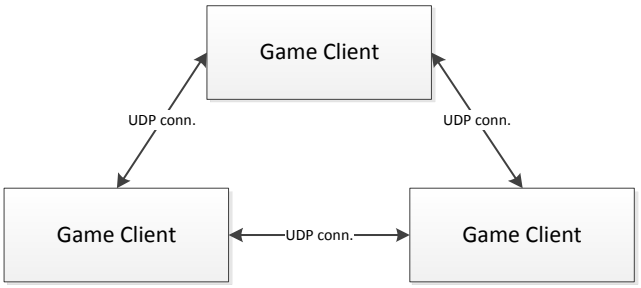


Figure 3.7: Illustration of GC communication.

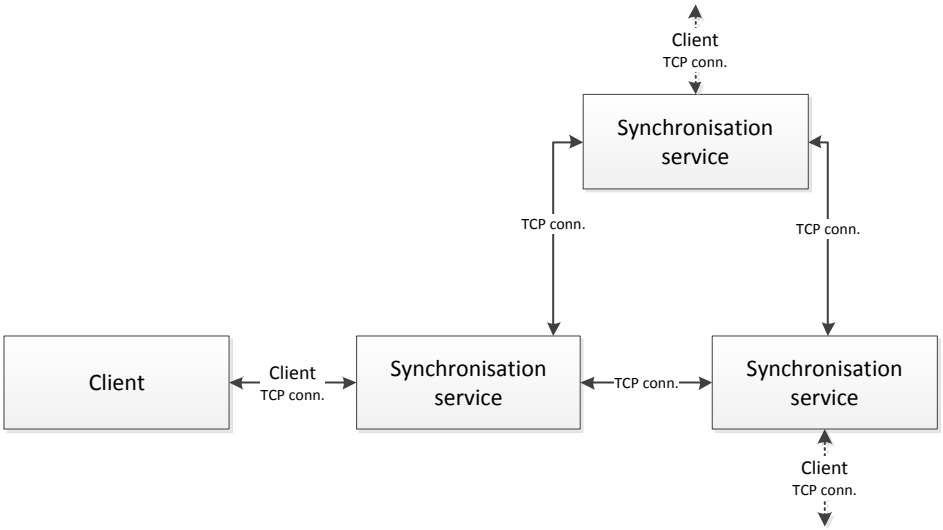


Figure 3.8: Illustration of SS connectivity.

to read and modify a variable requires a connection between reader and the owner of the variable. When a variable is updated the owner is responsible for broadcasting the variable's new state to all other peers.
Figure 3.8 shows connectivity between SSs and GCs and their independence from each other. The SS uses TCP for communication with the client and other SS.

3.5 Artificially Added Delay

In order to make the game fair, the game state must be as equal as possible across peers at the same absolute time. We propose a technique, that adds an artificial delay to events occurring in the game. A major problem with this is to determine the current latency between peers. This problem is discussed in Section 2.4.

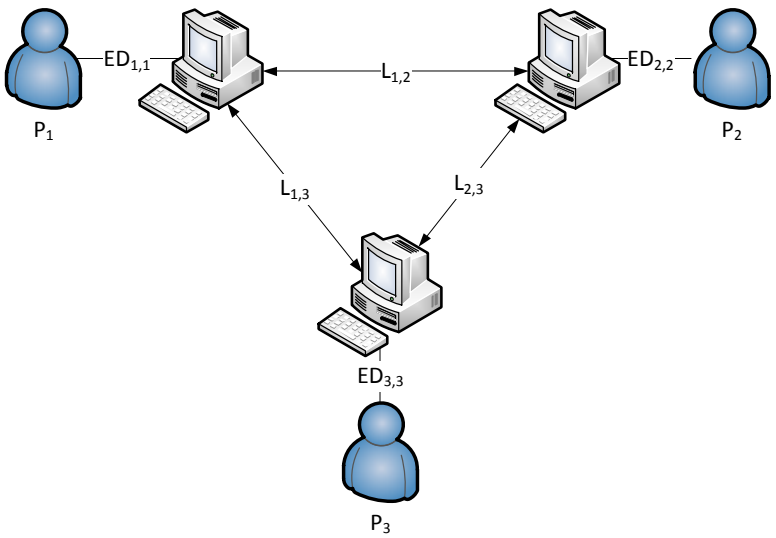


Figure 3.9: Illustration of users playing the game with all the variables which induce delay and where delay can be induced

The definitions below outlines the meaning of the variables used for discussion in this and related sections. The relationship between the variables P_x, $L_{x,y}$, $ED_{x,y}$ is shown in Figure 3.9.

Definition 3.4

Game delay (GD)**:** The delay in absolute time from the creation of an event untill the time it must be executed.

Definition 3.5

P_x**:** Player x, a participant in a game of Rawrlocks controlling avatar A_x.

Definition 3.6

$L_{x,y}$**:** Latency between P_x and P_y. This value is not adjustable as it is given by the network latency. We asumme that $L_{x,y} = L_{y,x}$, because we use RTT/2.

Definition 3.7

$ED_{x,y}$**:** The delay to be added for execution of P_y's events on P_x's screen. The amount of time events are postponed before being executed. An event is executed after a delay equal to GD on all peers from the absolute time the event was created. Thus, ED is calculated by using the following formula: $ED_{x,y} = GD - L_{x,y}$.

To illustrate the use of the variables, we present a simple example.

Example: A player performs an event, which should be executed at the same absolute time on all peers. In this situation $GD = 50$ms. The example is shown in Figure 3.10.

- P_1 performs an event at $t = 0$ms, and sends a packet to all other peers in the game.
- $L_{1,2} = 20$ms. The packet sent by P_1 arrives at P_2 at $t = 20$ms.
- $ED_{2,1}$ is: 50ms − 20ms = 30ms. Thus P_2 waits 30ms before executing the event of the received packet.
- P_1 waits GD before executing the event locally. Both players execute the event at $t = 50$ms absolute time after the event was created.

This means events contained in a packet are executed at the same absolute time if the latency between peers is less than GD.

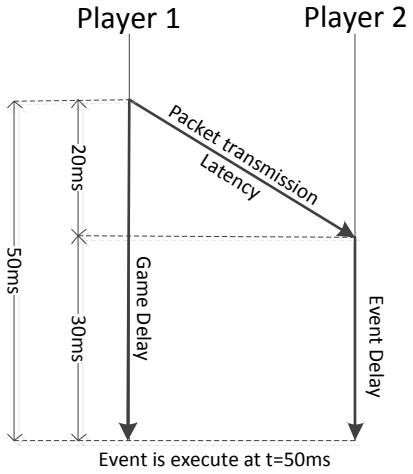


Figure 3.10: Situation where a player sends an event that is delayed to be executed at the same absolute time at two players.

In order to synchronise the events of the game, the game must be delayed by a certain amount of time. If all players in a game sessions should compete at an equal level, GD must be a value greater than or equal to the two players with the highest latency between them. The highest latency from a given player P_x to another player is defined as L_{max_x}

$$L_{max_x} = max(L_{x,1}, L_{x,2}, \ldots L_{x,n})$$

Thus if the players should compete at an equal level GD is determined as follows:

$$GD = max(L_{max_1}, L_{max_2}, \ldots, L_{max_n})$$

In doing this every peer seems to have the same latency. However, a pair of players with a high latency between them imposes an equally high delay to the rest of the players. Therefore, we define a limit, GD_{max}, which is the maximum amount of time the game can be delayed by. The revised formula is:

$$GD = min(max(L_{max_1}, L_{max_2}, \ldots, L_{max_n}), GD_{max})$$

One-way latency measurements must be carried out with reasonable intervals for this to work. More about a technique for this can be found in Section 2.4. We wish to keep a constant GD, so the formula used is:

$$GD = GD_{max}$$

Hence our solution causes a degradation in consistency as latency between peers increases above GD. A constant GD means that all local events are delayed by GD. The delay imposed by P_x on an event received from a remote player, P_y is:

$$ED_{x,y} = max(GD - L_{y,x}, 0)$$

This means that if $L_{y,x}$ is bigger than GD, events are activated instantly but delayed by latency. Therefore, a player where $L > GD$ to another player means event activation varies and this causes different game states across peers.

There are several solutions to keep a fast-paced game consistent. The two main categories are the ones using absolute packets and the ones using relative packets. Section 3.6 presents four solutions to artificially added delay. First of Section 3.6.1 and Section 3.6.2 presents two solutions using absolute packets. The solution presented in Section 3.6.1 uses delay to attempt to have the same event history played out on two players. However, movement events are not delayed. Section 3.6.2 presents a solution that delays events with a constant GD. In order to synchronise the position of avatars, peers distribute their location as it would be after GD. Thus, the position of every avatar is kept equal across clients at the same absolute time. Every other event in the game is being synchronised after GD. Next is Section 3.6.3 and Section 3.6.4 which presents two solutions using relative packets. The solution in Section 3.6.3 uses input-duration pairs. The pairs consist of an input key and a duration. Besides the pairs, the latest mouse position is contained in the packet. Section 3.6.4 presents a solution, that sends input on a per frame basis. This method uses the information of the latest frames to make up for packet loss.

3.6 Solutions to Latency Hiding

This section outlines the solutions we believe are relevant to accomplish latency hiding.

Any method to keep a game consistent are affected by latency. Latency has already been determined as a variable in Figure 3.9. It is important to note that latency is not measured exact. But the methods described here assume so. The reason why one-way latency is not exact is explained in Section 2.4. We define the maximum latency allowed for Rawrlocks as 50ms. This value is

henceforth known as *game latency*.
There are two ways to construct packets; absolute and relative packets.

Absolute Contains complete updates for game state objects, e.g. the position of Player 1 is now $(4, 6)$.

Relative Contains relative updates for game state objects, e.g. shift Player 1's current position by $(1, 0)$. The terms delta and relative are used interchangeably.

The packet system described in Section 4.2 can be used in the solutions with absolute packets. The solutions using relative packets requires a new packet system.

Mouse Position Mouse position is used to determine the direction for three abilities in Rawrlocks – namely fireball, thrust and teleport. therefore the position must be sent to peers. The position of the mouse can be delayed, making it consistent across peers, or the freshest mouse position can be used. No matter the implementation, the rendering of the mouse position must be with no added delay. This is because the mouse requires high precision and even a few ms delay is noticeable. It is the position used for ability calculations that can be artificially delayed. If it is delayed, it can be made identical across peers such that abilities are fired in the same direction. Players may be able to notice the delay, but tests are needed to prove whether or not this holds.

Another option is to send the freshest mouse position. By doing so, players should be unable to notice a delay in mouse position. However, the direction of a fired ability may vary across peers depending on the delay between peers and how the sending peer is moving his mouse. Flicking the mouse rapidly with high sensitivity gives the biggest difference on the local mouse position compared to the remote mouse position. Therefore, the direction of a fired ability is likely to have differ across every peer.

Without further ado we present each solution starting with the two using absolute packets.

3.6.1 Consistent Event History

This method seeks to ensure that situations play out the same way. The execution order of events is the same on all peers. A player's own events are locally delayed by *game latency*·2 except for local avatar movement. Events from remote players are delayed by *game latency*. The event can potentially be other events, e.g. power-up pickup. as long as it is only a single type of event This is because all other events have their time corrected in relation to the event that does not have any imposed delay. Movement is chosen as example in this discussion because it is easier to explain and because it makes sense to use an event where the user wants immediate feedback.
Since movement is not delayed locally, the local player P_x receives and executes a remote event from P_y after *game latency*. P_y execute's P_x movement event after *game latency* and his own event after *game latency*·2. Therefore the situation plays out in the same order on both players. This means two players'

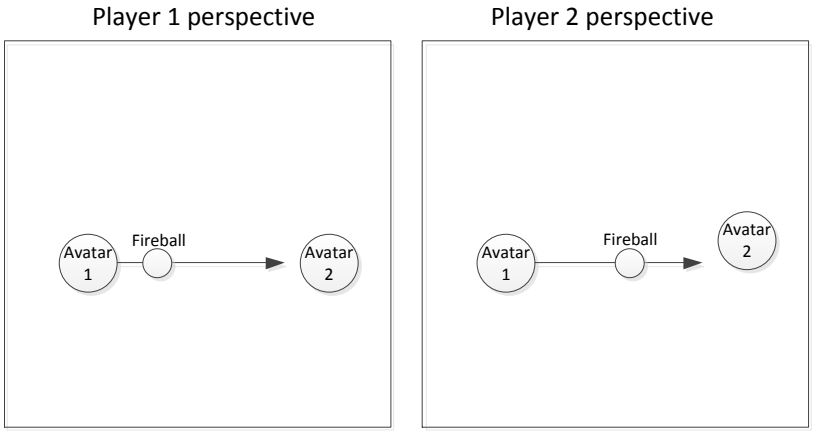


Figure 3.11: Illustration of users playing the game from two different perspectives at the same absolute time.

game state are never consistent at same absolute time, but events are executed consistently.

Example: In the situation where the latency between P_x and P_y: $L_{x,y} =$ 50ms. An event is delayed as follows:

$$ED_{y,x} = 0\text{ms}$$

$$ED_{x,y} = 0\text{ms}$$

$$ED_{x,x} = 100\text{ms}$$

These are applied to every event except movement where the event delays are as follows:

$$ED_{y,x} = 0\text{ms}$$

$$ED_{x,y} = 0\text{ms}$$

$$ED_{x,x} = 0\text{ms}$$

The first situation is P_1 firing a fireball towards P_2 with $L_{1,2} = 50ms$. The cast time for a fireball is 400ms. The scenario can be seen in Figure 3.11. The two perspectives are in the exact same absolute time and the situations are not equal, but plays out the same way if the game continues. Figure 3.12 illustrates when the events are executed in absolute time.

1. P_1 starts to cast a fireball towards P_2. It takes a total of 500ms before the fireball is fired on P_1's computer, due to the 400ms cast time and $ED_{1,1} = 100$ms for this event.

2. P_2 receives the event that P_1 is casting a fireball 50ms later in absolute time. P_2 commits that P_1 is casting the fireball to the local game state. In absolute time there is 100ms until P_1 commits the change to its local game state.

3. P_2 executes the fireball of P_1 in its local game 400ms after the event is received because $ED_{2,1} = 0$ms.

4. P_2 sends movement back to P_1 where P_2 avoids the fireball.

5. P_1 receives the movement 50ms after P_2 moved on P_2's computer. P_1 fires the fireball and sees that P_2 avoids the fireball. Thus, resulting in the same event execution history.

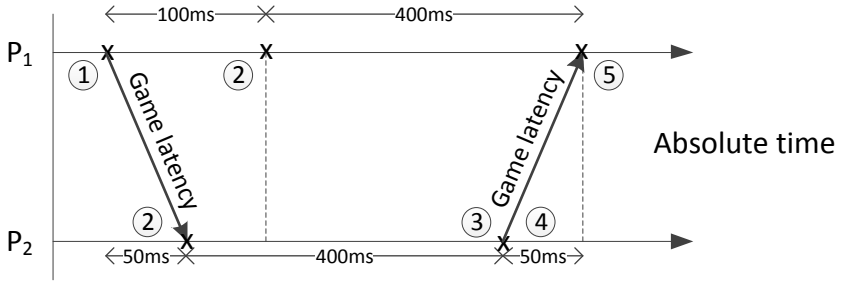


Figure 3.12: Illustration of event execution in absolute time. The circled numbers matches the numbers in the example.

The second situation is P_1 firing a fireball towards P_2 and P_3. The latency between peers is: $L_{1,2} = 20$ms, $L_{1,3} = 50$ms and $L_{2,3} = 50$ms. Figure 3.13 illustrates when the events are executed in absolute time.

1. P_1 starts to cast a fireball towards P_2 and P_3. It takes a total of 500ms before the fireball is fired on P_1'computer, due to the 400ms cast time and $ED_{1,1} = 100$ms for this event.

2. P_2 receives the event that P_1 is casting 20ms later in absolute time. P_2 waits $ED_{2,1} = \textit{game latency} - L1, 2 = 30$ms before committing the change to its local game state.

3. P_3 receives the event that P_1 is casting 50ms later in absolute time. P_3 commits the event to local game state.

4. P_2 and P_3 both moves to dodge the fireball. The movement is sent to P_1.

5. Both player's movement is committed to P_1 game state after 50ms.

6. P_2 movement is executed 50ms later at P_3 game state and vice versa. Thus both players' view of each other differs from P_1's view. However this is not that big of a problem because P_1 is the one who decides who is hit by the fireball.

Problems

This method gives consistent event execution at peers who creates an event at the cost of *game latency*·2. The downside is that participants does not see

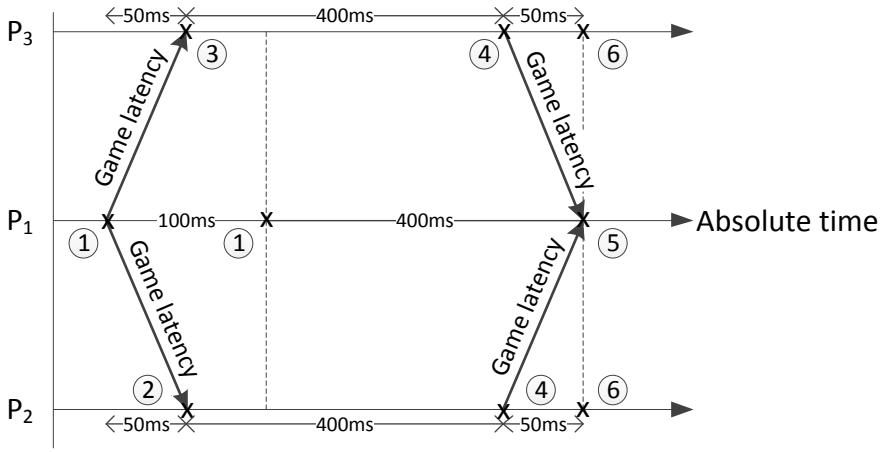


Figure 3.13: Illustration of event execution in absolute time. The circled numbers matches the numbers in the example.

each others movement in the same absolute time. Movement is delayed by *game latency* before they are committed to the remote players' game state. The main problem with this method is that all events besides movement is delayed by 100ms because the delay is inconsistent across events. Whether or not people notices that movement is not delayed while every other event in the game is delayed is unknown. Furthermore, events are not happening at the same absolute time across clients. Meaning, that the game state is never consistent across peer in an active game.

Another problem is that the model can only handle half the network latency the other models can handle because events take twice the time to execute and players notice too much added latency.

3.6.2 Position Prediction

This solution uses absolute packets and a technique to predict an avatar's position. The solution seeks to keep the game consistent by using GD

The position is predicted locally before a packet is sent and works as follows:

$$Pos(P_{local}, t + GD) = Pos(P_{local}, t) + (\hat{V}_P \cdot S_P + KB_P) \cdot GD$$

t is the current time, $Pos(P_{local}, t + GD)$ is the position of player P_{local} at time $t + GD$, $\hat{V}_P$ is the normalised direction of the player's velocity, S_P is the current speed of the player and KB_P is the player's knockback vector.

The value of GD can be as low as the *game latency*. Activating events at the same absolute time across peers should make the game state equal across all peers. However, this solution requires exact knowledge of the latencies between peers.

3.6.3 Input-Duration Pairs

This solution uses relative updates and seeks to activate keys at the same absolute time across peers. The idea is to emulate the same gameplay across peers by sending activated keys.

The packets in this solution consist of a list of pairs. A pair consists of an input key and the current duration the key has been pressed. The pairs are henceforth known as input-duration pairs. The information in the packets is used to make relative game state updates. Besides the pairs, a packet contains the newest mouse position.

A packet contains the nine most recent input-duration pairs and one mouse position.

Whenever a key is pressed one pair is set; the key is set to that of the pressed key and the duration starts incrementing. When the key is released the pair consists of a key identifier and a duration. Pairs are set the moment a key is pressed. A pair is finalised once the key of the pair is released or if a key is pressed for more than 256 frames, in both cases a new pair is instantiated. If two keys are pressed at the same time, two pairs are set. The second key press activates the second pair and so on. This means that pairs are in a list with a maximum length of nine pairs. When a new pair is inserted the oldest pair is pushed out.

Receiving Input-Duration Pair Updates

Input-pairs are delayed by 50ms before being activated for the duration on both sending and receiving peer. The duration value of the pair is updated whenever a value, that is higher than the previous value, is received. If the duration value received is equal two times in a row, the player sending the pair has released the key in question.

When a movement input is sent it contains directions instead of keys. The players can send nine different directions – one for each direction of which one is the zero vector i.e. standing still. Each movement update is stored and converted to an in-game distance that is added to a locally stored player position.

Problems

Ability activation and ability hits are not guaranteed to be fired from the same position or with the same mouse position. Because this solution uses relative updates the game is likely to become inconsistent across clients, since lost updates are not re-sent. Peers do not send events, so whether peers agree on an ability hitting a target depends on latency, packet loss and if mouse position is delayed. In order to have the same game state across peers, input must be registered in the same order at the same frame across peers.

3.6.4 Sending Input Per Frame

This solution uses relative updates. It seeks to update the game state every 50ms and each iteration is called a frame. The game is delayed by 50ms. Each player sends input on a per frame basis. The information sent could be as follows:

Frame number n
4 keys for frame n
mouse position for frame n
4 keys for frame $n-1$
mouse position for frame $n-1$
4 keys for frame $n-2$
mouse position for frame $n-2$
4 keys for frame $n-3$
mouse position for frame $n-3$

The input for the four latest frames is sent to every peer. The current displayed frame is $n-1$ given a latency of 0ms. This means that the input for frame $n-2$ is too old when it is received even with a latency of 0ms. Therefore, the frame information of the three oldest frames can be used to readjust the position of the player sending the packet in case of packet loss.

Starting the Game

When a frame n is ready to be computed, frame $n+1$ is sent to each of the other peers. This means that in order to proceed to the next frame, a peer needs packet information for the current frame from every other peer.
When the game starts the first two packets are calculated as soon as possible i.e. at 0ms and after 50ms. Each peer sends the packets to all the other peers. When a peer has received information for frame 1 from every other peer, the game state of frame 1 is calculated and rendered. This happens no sooner than 50ms in absolute time after the frame input. The Input of frame 2 is calculated and sent to each peer. When frame 2 is received, frame 3 is calculated and distributed. This happens in intervals of approximately 50ms.

Having consistent input for each frame ensures that events progress in the same order. However, the input must be received no later than 50ms after being activated. Therefore, the game should be consistent if there is no packet loss.

Problems

If a packet is lost, the remote player does not know the input for the next frame. A player knows that a packet has been lost if the difference between two concurrent frame numbers is more than 1.
Two methods can be used to prevent the effects of packet loss:

1. The input in two consecutive frames is likely to be similar. If input for a frame is unknown, the last known packet is used.
2. If a packet is not received at the time it is needed, the position of the player sending the input can be readjusted when the packet arrives.

If a peer loses four packets in a row, the input of the peer is never received and the game state is likely to become inconsistent.

3.6.5 Conclusion

The solutions using relative updates are likely to become inconsistent in case of packet loss or out of order packets. I.e. a fireball could be fired a frame later on one client's computer, which results in a player avoiding a fireball he should not have avoided. Further updates from this situation results in the game getting more and more inconsistent, and a consistent game state is never again reached. The *consistent event history* solution suffers from the problem that two players do not play out the same event at the same absolute time. The game state is never equal on two peers at the same absolute time while the game state is being altered. Thus, one can argue that the game state is never consistent. The solution using position prediction has the problem that if an avatar changes direction between two frames other peers must be notified as fast as possible. Since all events are executed after the same delay, the game state stays consistent on all peers.
Common for all solutions is that the difference between game states increases concurrently as latency goes above GD. In order to have a viable solution that can be tested against our highlighted problems as described in Section 1.1, we have decided to use the solution described in Section 3.6.2. This solution has a consistent game state across peers and uses absolute packets.

4 Implementation

This chapter describes the implementation of the services required to keep Rawrlocks consistent. In Section 4.1 the SS implementation and, specifically, the SS packet structure is described. Afterwards, Section 4.2 describes the implementation of the GC with a close look at the GC packet structure.

In Section 4.3 we describe the implementation of the chosen latency hiding solution as described in Section 3.6. Lastly, we conclude on the implementation in Section 4.4.

4.1 Rawrlocks Synchronisation Service Packets

Due to the constraints defined in Section 3.4.2 the packets sent via the SS are constructed simple. The structure of packets sent between SSs is shown in Table 4.1(b). The figure contains three attributes that are described below:

Packet Type Modify update or Set update.

Variable ID ID of the variable maps to the same variable name on all the SS. This is because game clients initialise the variables in the same order on all SSs.

Value The value to modify or set the variable with.

Packets contain an IPv4 header and a TCP header plus data and costs $20 + 20 + 6 = 46$bytes. The TCP stack automatically combines multiple packets sent at the same time to the same destination. The worst case situation with 10 players is a player picking up a power-up. The result is two updates sent to every player with a burst of $((20 + 20) + 6 \cdot 2) \cdot 9 = 468$bytes.

4.2 Rawrlocks Game Client

This section describes the GC packet implementation in Rawrlocks. In Section 3.3, we analysed the variables of Rawrlocks in order to find the ones that required synchronisation. The variables that requires distribution via the GC are:

- Avatar Specifics
- Ability Cast
- Ability Fire
- Ability Hit

Furthermore, we defined the freshness and exactness requirements for each variable, and whether the event is triggered or activated.

A packet in Rawrlocks consists of these four variables and a prefix with a 1 byte hash of the string "rawrlocks". The hash is checked upon receiving a packet and the packet is discarded if the hash is wrong. While the freshness requirement is high for all variables in a packet the exactness requirement varies. The requirements for each variable is shown in Figure 3.3. Due to the high exactness requirement on triggered events, a loose form of reliability is required as described in Section 3.3. In order to distribute as little information as possible, we analyse which values are required for the triggered events:

Ability Fire

In order to activate an *Ability Fire* event two things are required:

1. The position it is fired from and in case it is a fireball, the direction it is fired in.
2. The ability type.

Number 1 is updated independently of the *Ability Fire* event. Therefore, this information can be omitted when updating *Ability Fire*. Number 2 is specific to the event and must be known in order to activate the correct event.

Ability Hit

In order to activate an *Ability Hit* event two things are required:

1. Which of the recently fired abilitis is the one hitting
2. The targets of the ability.

Once the fired ability is known, the impact specific details, e.g. knockback, are retrieved from the fired ability and is applied to the targets. If the correct fired ability is not found, the avatar may have erroneous impact details applied, but the game state does not reach an inconsistent state because the clients synchronizes the inconsistent state and it therefore becomes the consistent state.

Triggered Event Reliability

Triggered events in Rawrlocks need only a loose form of reliability. Neither of the triggered events update game state variables with high exactness requirements. Thus, if a triggered event is lost, the game state remains consistent. However, losing a single packet should not result in the player not being able to see whether a triggered event occured. The information required to execute the latest triggered event is sent with every packet. If a packet is lost, an event can still be executed using the next packet.

In order to achieve the desired level of reliability, events are prefixed with a 4 bit sequence ID. Whenever a triggered event occurs the sequence ID increments. The packet is updated with the event information required to activate the event and the packet is sent to every peer.
The sequence ID loop, such that if the last received event sequence ID is above 13, a sequence ID of less than 6 is considered an increment.

4.2.1 Rawrlocks Game Client Packets

The structure of a Rawrlocks packet is as depicted in Table 4.1(a)

Avatar Specifics

This part of the packet contains general information about the avatar. Who controls it, where it is, the direction it is moving and its knockback.

Avatar Cast

When an avatar casts an ability, its appearance changes in the form of a color change. This part contains a value indicating whether to cast or not, and a value indicating which ability to cast. Thus, while a player is not casting, the value is still being sent with each packet.

Ability Fire

When an ability is fired the type of the fired ability is sent. Whenever an avatar receives an ability fire event, the fired ability is added to a list of fired abilities unique to each avatar. The ability is placed in an array at the position equal to that of the received counter.

Ability Hit

In the event of an avatar being hit by an ability, the player who initiated the ability makes every other participant aware of the event. This part contains an ID of the fired ability that hit and the targets hit. Whenever an avatar receives an ability hit event, the fired ability is fetched from the avatar's array of fired abilities. The specific impact details are calculated from the specifications of the fired ability.

Table 4.1: The structure of the packets sent between nodes. Table 4.1(a) is the GC packets while Table 4.1(b) is the SS packets.

(a)

Game Client Packet Structure							
Description	Packet Parts	Rawrlocks hash	Avatar Specifics	Ability Cast	Ability Fire	Ability Hit	Total
Size in bytes		1	14	1	2	3	20
Packet Parts							
Description	Avatar Specifics	Avatar ID	Position	Move direction	Mouse position	Knockback	Total
Size in bytes		1	8	1	2	2	14
Description	Ability Cast	Casting	Skill Type				Total
Size in bytes		1/8	7/8				1
Description	Ability Fire	Sequence ID	Skill Type				Total
Size in bytes		1/2	1/2				1
Description	Ability Hit	Sequence ID	Ability Hit ID	Targets			Total
Size in bytes		1/2	1/2	2			3

(b)

Synchronisation Service Packet Structure				
Description	Packet Type	Variable ID	Value	Total
Size in bytes	1	1	4	6

4.2.2 Packet Conversion

A Rawrlocks packet can be converted to an array of bytes and vice versa. In order to convert we use some data compression methods to make the packet size as small as possible.

Direction A direction is compressed from a 2D vector to a number between 0 and 255. The number 255 is the zero vector.

Small numbers Two small positive integers (between 0 and 15) are represented as one byte such that the first half of the byte is one number while the other half is the other number. This method is used for sequence ID's and skill type.

Booleans Boolean values are added together with a small positive integer such that the first 7 bits are used for the number and the last bit is the boolean.

Distance The distance to the mouse cursor from the avatar. A number between 0 and 90000 is converted to a number between 0 and 255

Knockback The length of the knockback vector is a number between 0 and 600. This is converted into a number between 0 and 255 so it can be used as a byte. The knockback direction is converted as every other direction

Each conversion has a method to convert to and from a byte. When a packet object is converted to a byte array each variable is converted to a byte or an array of bytes. The bytes of the packet parts is joined to a byte array. The byte arrays of each packet part is joined to form a byte array that contains the packet information. For the reverse conversion the byte array is read and the value for each variable in each part is set.

4.2.3 Packet Creation

When the game is initialised a local packet is created. The avatar ID is set to that of the local player's ID.
Each of the packet parts contain a function to update the data of the packet part from an event. Thus, whenever an event is queued the local packet is updated accordingly. See Section 4.3 for information on events and event queue.

4.2.4 Packet Handling

When a packet is received the replicated avatar corresponding to the received avatar ID is extracted from the local avatar list. New events are added to the avatar's event queue as per the data in the packet. Determining the position of the avatar is described in Section 4.3.2. *Ability Cast* is handled straight forward. If the value indicates that a player is casting, the skill-type determines which ability the avatar is casting. The color of the avatar is changed accordingly.
For *Ability Fire* it is checked whether the sequence ID is newer than the last

ID. If it is, the ID is updated to the new sequence ID and the projectile ID and ability type is stored in an array at a position equal to the sequence ID. The ability contained in the packet is fired from the sending avatar's current position and direction.
The *Ability Hit* is handled in a similar way. First, a check to see whether the sequence ID is higher than the latest ID. If this is the case, the ID is updated and the ability data is gathered from the list of fired abilities. The target of the ability applies knockback if the target is the local player. The knockback depends on the ability, which is described in Section 2.5. Players affected by knockback sends it as part of their packet.

4.2.5 Packet Loss

The game is designed to handle packet loss. However, the effect of packet loss varies depending on the data contained in the lost packet. If there are no changes to triggered events and the avatar does not change direction, packet loss is unnoticeable. This is because the position of remote avatars is extrapolated if no new positions are known. However, if the first packet containing a triggered event is lost, the event has further delay imposed, and thus the time difference between activation across peers is bigger than if no packets were lost. If a peer loses all packets for a second or two, the peer is likely to lose a triggered event, meaning that either an ability is not fired on the peer's client or a hit does not occur. We argued, in Section 3.3, that the loss of a triggered event should not cause the game to reach an inconsistent state.

4.3 Event Delay

Section 3.5 discusses solutions to add artificial delay to events. This section contains an explanation of how the selected solution, presented in Section 3.6.2, is implemented in Rawrlocks.
There are five types of events in Rawrlocks that are distributed via the GC. These events require the following information updated:

Cast Event Cast events contain two values:

- A boolean determining whether or not to cast.
- An integer determining which ability to cast.

Fire Event Fire events contain one value: Which ability to fire.

Hit event Hit events contain two values:

- A value determining which of the abilities that was fired that hit.
- Which enemies are hit.

Knockback event In order to activate a knockback event, the knockback vector must be known.

Move event The direction an avatar walks.

These events are the ones necessary to alter the variables distributed via the GC decided in Section 3.3.

4.3.1 Event Base Class

Each event inherits from the event base class, that contains a set of common values and functions:

Owner The avatar that owns the specific event. The owner can either be local (controlled by the player in front of the computer) or remote.

Condition The *condition* of the event. If the *condition* is true, the event must be executed. This value is only used for local events.

Event Queue A queue of events specific to the owning avatar and the event type. The data contained in an element in the queue is described in Section 4.3.

AddToEventQueue Adds an event to the event's *event queue*.

AddToLocalPacket Adds the data of the event to the next packet to be sent.

Activate Executes the event.

The events run differently depending on whether the owner is the local player or a remote player. Figure 4.1 is a flowchart diagram, that illustrates what happens at the local player, P_{local} (top half of the figure), and remote player, P_{remote} (bottom half of the figure), in the event algorithm.

Local Player Events

If *condition* is true, two things happen in P_{local}'s GC:

- The event is added to the *event queue* via the *AddToEventQueue* function.
- The data of the event is added to the local packet via the *AddToLocal-Packet* function.

If the event is a triggered event, the local packet is sent to all remote peers immediately. Otherwise, packets are sent at the regular rate of $(n-1) \times 20$ packets per second to remote peers.
When an event has been in the *event queue* for GD the event is activated via the *Activate* function and removed from the event queue.

Remote Player Events

The local GC receives a packet from P_{remote} after a delay of $L_{remote,local}$. The packet contains the data of an event. The local GC uses this data to create and add the event to P_{remote}'s *event queue*. An event from P_{remote} is executed at P_{local} after $ED_{remote,local} = GD - L_{remote,local}$ via the *Activate* function. After activation, the event is removed from P_{remote}'s *event queue*.

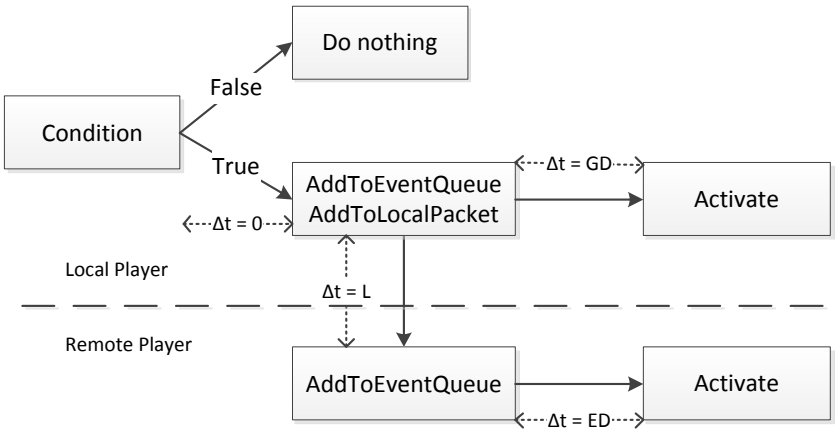


Figure 4.1: Local- and remote player event activation.

4.3.2 Position Prediction

As per the design of our latency solution described in Section 3.6.2, the position of the local avatar is predicted before it is sent.

In order to meet the freshness requirements of avatar position, a change in direction triggers a new packet, that is sent instantly to every other peer. When a player receives a packet, a move event is created with the received position. When the event is activated, the position of the player is set to the received value.

Position Prediction Between Received Points

For every remote player in the game the three latest received positions are stored. t_n is the time when the newest received position is active and t_o is the time when the oldest received position is active. If $t_n > t$, P_{remote} moves from its current position to the newest received position.
If $t_n < t$ we extrapolate in a line from $Pos(P_{remote}, t_o)$ to $Pos(P_{remote}, t_n)$.
If $t_n = t$ the position of the avatar is set to the received value.

The three situations are listed below:

If $t_n > t$:

$$Pos(P_{remote}, t) = Pos(P_{remote}, t) + (Pos(P_{remote}, \widehat{t_n) - Pos(P_{remote}, t)}) \times S_P$$

If $t_n < t$:

$$Pos(P_{remote}, t) = Pos(P_{remote}, t) + (Pos(P_{remote}, \widehat{t_n) - Pos(P_{remote}, t_o)}) \times S_P$$

If $t_n = t$:

$$Pos(P_{remote}, t) = Pos(P_{remote}, t_n)$$

Using these formulas we can calculate the position of an avatar at time t, whether or not a new value of the position of the avatar has been received.

4.4 Implementation Issues

This section describes some of the issues regarding the implementation of Rawrlocks.

4.4.1 Mouse Position Delay

In the current implementation, the mouse position is not delayed. This means that whenever P_{local} receives a packet from P_{remote}, the mouse position of P_{remote} is set when it is received. Therefore, the mouse position of players is not consistent and so, fire events are likely to be activated in different directions across peers.
E.g: A situation where $L_{remote,local} = GD$. P_{local} activates a fire event. At $t + 0$, P_{local} sends the fire event and direction to every other peer. At $t + GD$, P_{local} activates the fire event in $Dir(P_{local}, t + GD)$. However, remote peer activates the fire event in $Dir(P_{local}, t)$, since it is the newest received direction of P_{local}.
The amount mouse direction can differ across peers depends on the latency between peers and how fast P_{local} moves his mouse. If P_{local} moves his mouse from one side of his avatar to the other, the direction changes 180°. The worst case scenario in the current implementation is that a fire event is activated in opposite directions, which is an unacceptable situation. In the current implementation mouse position is delayed by $L_{remote,local}$ for remote players due to the latency.

If we were to delay mouse position, players might be able to notice the imposed delay. The implementation decide the direction of a fire event when it is added to the event queue instead of when it is activated. If fire events were handled this way, the fire events would be activated in the same direction across clients if $L_{remote,local} \leq GD$ and the first packet containing the fire event is not lost.

4.4.2 Packet Reordering

Currently, packets do not contain a sequence ID, and thus, there is no method to detect if a packet is out of order. If P_{local} receives an old packet from P_{remote}, P_{local} mistakenly believes the delay of the packet is $L_{remote,local}$ while the packet in reality is further delayed. Therefore, the position of P_{remote} is set to an incorrect value upon activating the misordered packet. Packet reordering can be avoided if a packet sequence ID is introduced. Thus, if a packet has wrong ordering, the packet sequence ID is lower than the newest received packet sequence ID.

4.4.3 Latency Estimation

Our latency estimation technique is simple: $RTT/2 = latency$. Other latency estimation techniques use the same approach. However, Choi and Yoo [15] proved that $latency \neq RTT/2$. Latency estimation using RTT differs by up to 25% from the real latency. Thus, if $latency = 50$ms then the estimated latency varies with ± 12.5. This is a value far below what humans are able to detect. Furthermore, other games have used $RTT/2$ as an estimation of latency successfully.

5 Testing

This chapter presents the testing conducted on Rawrlocks. Section 5.1 describes the environment used to run our tests. Besides the description of the test environment the section contains an explanation of the global timer that is used to compare results and how the network between the test environment computers is emulated. Section 5.2 contains descriptions of the scenarios used to test how consistent Rawrlocks is along with a discusion of the results. Section 5.3 describes the bandwidth testing, the results are used to determine how much traffic Rawrlocks produces.

5.1 Environment

The test environment is the set of computers we use to test Rawrlocks. The environment consists of four identical *Dell Optiplex GX620* computers and two *Lenovo SL500 2746* notebooks connected via ethernet through a *D-Link DES-1008D* switch.
The specifications are as follows:

- Dell Optiplex GX620: Intel Pentium D 820 2.8GHz, 1GB DDR2 memory, ATI Mobility Radeon X600.
- SL500 2746: Intel Core 2 Duo 2.0GHz(AQG) and 2.4GHz(49G), 2GB DDR2 memory, NVIDIA GeForce 9300M GS 256MB
- D-Link DES-1008D: 8 10/100mb ports.

Computer 1-4 are Dell Optiplex. *Computer 5-6* are Lenovo SL500. The computers have the following setup and roles in testing:

Computer 1 is running Ubuntu Linux 10.10. The computer has *netem* installed.

Computer 2-6 is running Windows 7 and plays the game.

A test runs as follows: *Computer 1* starts the game, kills the game and collects the log files from *Computer 2* to *Computer 6*. The consistency tests run on *Computer 2* and *Computer 3* or *Computer 5* and *Computer 6*. The bandwidth tests run on all the game computers, *Computer 2-6*.

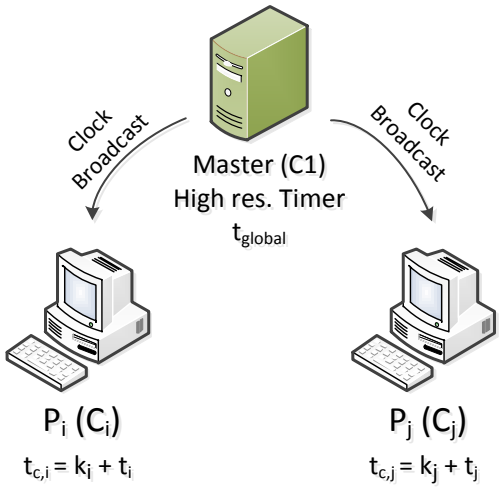


Figure 5.1: The master (*Computer 1*) has a high resolution timer. The timer is broadcasted to peers to synchronise timestamps in logs. In this figure x is replaced with the respective peer i, j.

5.1.1 Global Timer

In order to test whether events execute at the same absolute time, we have implemented a global timer.
Computer 1 runs a high resolution timer [25] and broadcasts the current time using an UDP packet with destination *224.100.0.1* with a 100ms interval. The broadcast is shown in Figure 5.1. The time broadcasted is known as t_{global}. The packet is not broadcasted onto the subnets used for test cases, but onto the normal network which has no induced latency or bandwidth restrictions. After starting Rawrlocks *Computer 2-6* receives the first packet from *Computer 1*, *Computer 2-6* sets a local value k_x such that $k_x = t_{global}$. Furthermore, *Computer 2-6* starts a local high resolution timer, t_x, that counts up from 0. The current time on *Computer x* is $t_{c,x} = k_x + t_x$.

Every five seconds *Computer 2-6* listens for a new value of t_{global}. Upon receiving a new value of t_{global}, the value of k_x is updated such that $k_x = t_{global} - t_x$ Due to local network delay and CPU scheduling we accept $t_{c,x}$ to drift $\pm$5ms from the value of t_{global}. If a test contains a time difference bigger than 5ms the test is discarded.

5.1.2 netem

In order to test Rawrlocks with varying latencies, we use *netem*, a module that provides a network emulation functionality [26]. *netem* can, amongst a list of filters, add a network latency filter. The network latency filter can add a specific delay to incoming and outgoing packets.

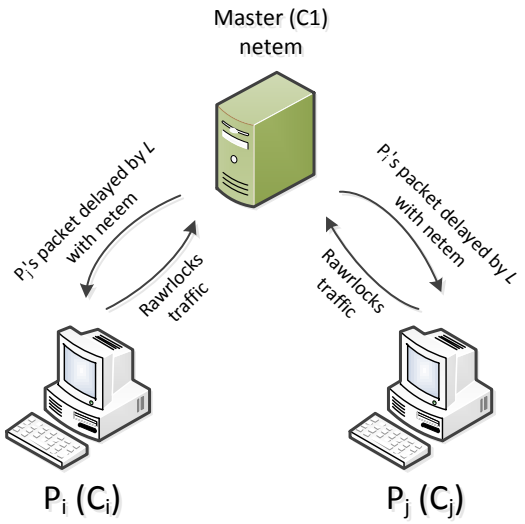


Figure 5.2: Setup with netem on the master computer.

Testing with netem

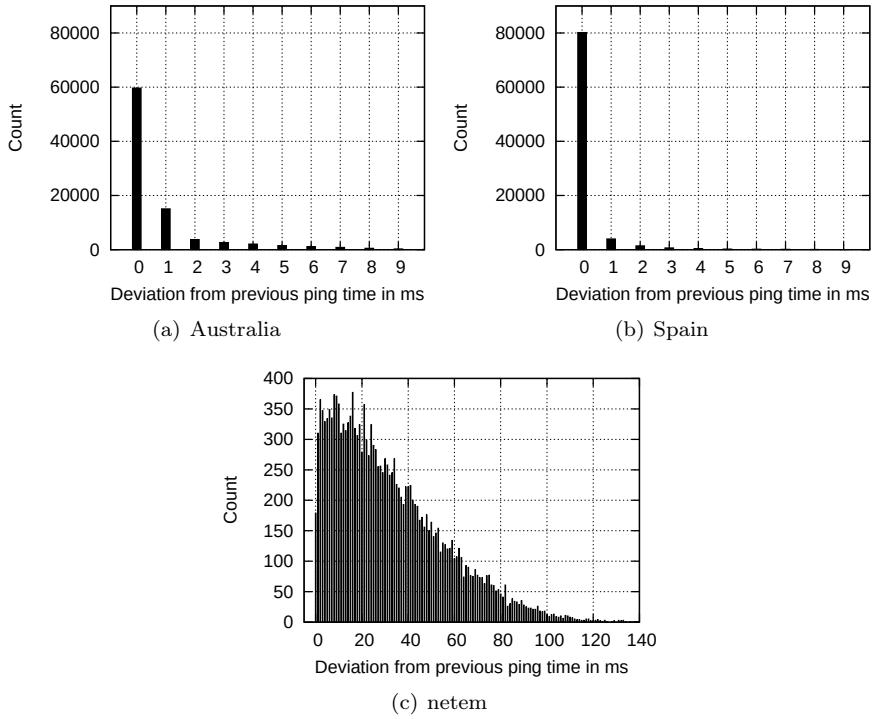
Setup Our test setup is illustrated in Figure 5.2. *Computer 1* has *netem* installed. All traffic created by Rawrlocks is routed through *Computer 1* before it is arrives at the receiving peer. *Computer 1* adds a static latency to outgoing packets.

Latency To test variable added latency with netem, we need real world samples to compare with. In order to test latency stability in a real world internet environment, we ping a server in Spain and a server in Australia from a server located in AAU's network ≈ 87600 times at a rate of one time per second. The results are shown in Figure 5.3(a) and 5.3(b). The figure shows how much ping times deviates from the previous ping. The majority of pings do not deviate from ping to ping. Combined $\approx 2.1\%$ deviates with 6ms or more. The maximum deviation for Australia is 14ms and 212ms for Spain, but these deviations happened once or twice over a 24 hour period.

In order to test the artificial latency distribution of *netem*, we add a delay filter with a normal distribution on *Computer 1*. The delay is 50ms and it is varying with $\pm$20ms. Thereafter, we ping *Computer 2* from *Computer 3* while routing the traffic of *Computer 3* through *Computer 1*. The distribution graph of the test is shown in Figure 5.3(c). While the real world tests show $\approx 2.1\%$ with 6ms or higher deviation, *netem* has $\approx 90\%$ over.

While it is possible to add our own distributions based on experimental data in *netem* it is beyond the scope of this project. Therefore, we do not test with varying latency but use a predefined fixed latency.

Packet Loss The game is designed to handle some degree of packet loss. If a packet is lost in Rawrlocks, it does not need to be resent. The information in the lost packet is still contained in the following packet provided that no



(a) Australia

(b) Spain

(c) netem

Figure 5.3: Ping deviation between each ping, by pinging every second.

new triggered events are activated. Due to time constraints we do not test for packet loss.

5.1.3 Test Scenario

A test runs like this:

- *Computer 1* contacts the computers playing the game and informs them to start the game.
- The game runs for a predefined time span. This span varies and depends on how much time it takes for one test run to complete.
- *Computer 1* synchronises $t_{c,x}$ of the computers. One of the computers playing the game is chosen by the SS to decide a test start time. The starting time is relative to t_{global}. When each players' $t_{c,x}$ reaches the specified value, the computers start the test locally. It is assumed that the test is started at the same absolute time across peers.
- After a test has run for the specified amount of time, *Computer 1* tells the other computers to kill the game and collects the logs.

This process loops until a sufficient amount of logs are gathered.

	25ms	50ms	100ms	250 ms
with ED	×	×	○	○
without ED	○	○	÷	÷

Table 5.1: The types of test cases we have chosen to run for our consistency tests. "×" is a test inside our requirements while "○" is outside our requirements. "÷" means the test is not performed.

5.2 Consistency Testing

The tests described in this section help determine if Rawrlocks stays consistent between two peers. We construct two different scenarios, each following a script. By using the same scripts we can replay the same situation several times, and check if the participants agree on the outcome.

Each scenario is tested several times with minor modifications in order to test slightly various situations and compare the results. In the tests with event delay, events are delayed by a value of 50ms. The tests are conducted without graphics rendering and run at $\approx$ 16000 FPS.

Table 5.1 provides an overview of the different test cases carried out in this section. In the table it is depicted which test cases are inside and which test cases are outside our requirements. A test inside our requirements means that the test is run within our set latency limit of 50ms and with the implemented event delay. Outside our requirements means that the test is run either with a latency above our latency limit or without event delay.

The following sections provide a thorough description of the scenarios.

5.2.1 Scenario 1: Fireball

This scenario has two actors; avatar 1, who moves in a straight line and avatar 2, who shoots a fireball in a vector perpendicular to avatar 1's movement vector. The scenario is illustrated in Figure 5.4. A maximum of one fireball may hit avatar 1 because the cooldown on the fireball ability is bigger than the amount of time it takes for avatar 1 to walk across the area where he may get hit by a fireball.

Variables We modify this scenario in two ways;

- Change the initial position of avatar 1 – depicted in Figure 5.4 by the *adjustment vector*.
- Change the network latency between the peers.

By changing the initial position of avatar 1 we get situations where the fireball hits avatar 1 and situations where the fireball does not hit. By testing with latencies within our requirements (25ms and 50ms) and latencies above (100ms

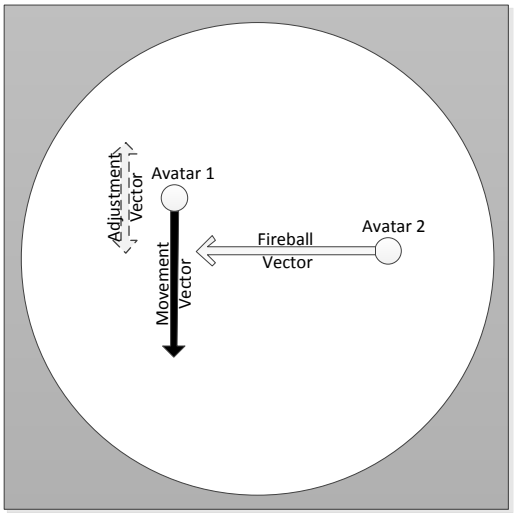


Figure 5.4: Illustration of fireball scenario. Avatar 1 moves along a vector while avatar 2 fires a fireball in a vector perpendicular to avatar 1's movement vector

and 250ms), we see if Rawrlocks stays consistent both within and outside our requirements.

Motivation This test is conducted to check three aspects of the Rawrlocks implementation:

- Do the peers activate the ability-fire events at the same time. The time we compare is $t_{A-fire1}$ and $t_{A-fire2}$ in Figure 5.5. This metric tests our event delay.
- Do the peers agree on the game state with regards to ability-hit events. This metric tests if our latency hiding technique can be used to calculate future game states.
- Are the positions of the avatars equal across peers when an ability-hit event occurs? This metric tests our position prediction.

Measurements In this test we are interested in three metrics:

Event execution test We look at the time ability-fire events are activated locally for both players. We display the time differences between event activations on a chart.

Agreement test We look at whether both players agree on a fireball hit or not. The test results list the number of hits the players agreed on and the number of hits they disagreed on.

Position prediction test We look at the position of the player being hit by a fireball when the ability-hit event is activated. Both nodes log the

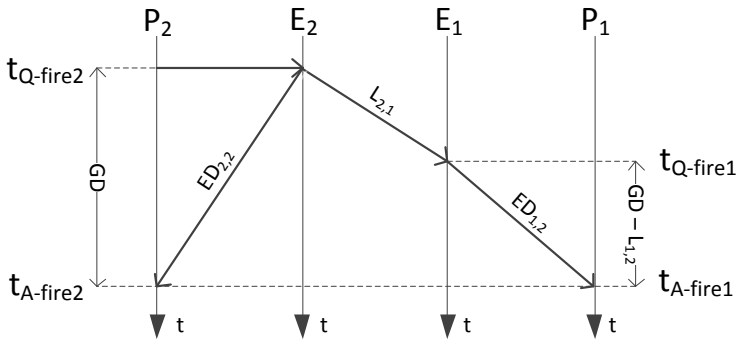


Figure 5.5: Illustration of the time we use to compare fireball event activation. Player 2, P_2, queues a fireball ability-fire event at time $t_{Q-fire2}$ and his engine, E_2, sends the event to E_1. The ability-fire event is activated at P_2 after GD at time $t_{A-fire2}$. After the event is received at E_1 the event activates at P_1 after $GD - L_{2,1}$ at time $t_{A-fire1}$. The compared times are $t_{A-fire1}$ and $t_{A-fire2}$.

position of the avatar being hit when the ability-hit event is activated. We calculate the maximum and average distance between player position at ability-hit event activation. Note that if the ability-hit event is not activated at the same time, the position of the avatar is not logged at the same time either.

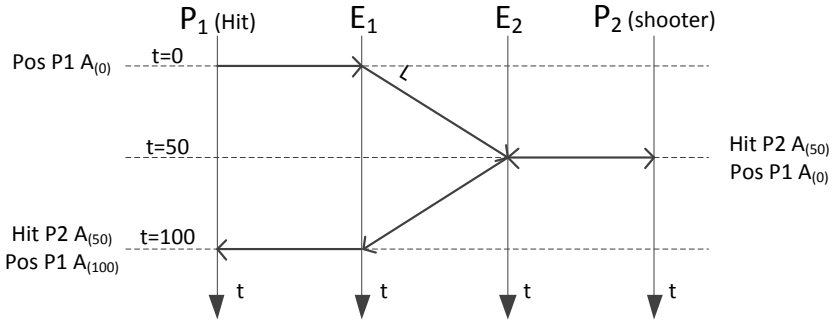
The tests satisfy our goals by the following metrics:

Agreement test An agreement test satisfies if the players agree on ability-hit events in at least 99% of the tests.

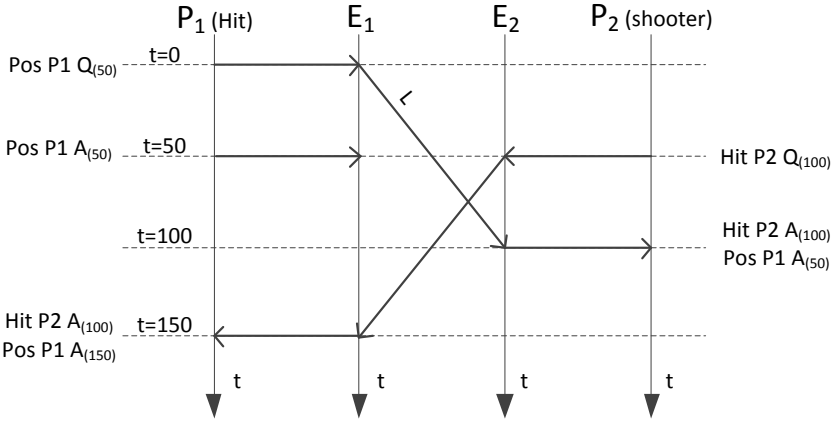
Event execution test An event execution test satisfies if the events activate within ±10ms, out of which 5ms stems from clock synchronisation inaccuracy.

Position prediction test A position prediction test satisfies if the position of avatars vary with less than one yard. For reference, the avatar has a radius of 23 yards.

Expected Outcome We expect the tests to satisfy our requirements if the latency is less than or equal to GD and event delay is activated. Otherwise, we expect the tests to fail. Furthermore, we expect to see similar results for tests with 100ms latency with event delay and tests with 50ms latency without event delay. With regards to position prediction, the situations are depicted in Figure 5.6. The values to be noticed in the figure are what the time of the position activation is when the ability-hit event is activated. For both figures the difference in time of position activation when the ability-hit event is activated is 100ms. An avatar can move 9 yards in 100ms, and thus we expect the *position prediction* "100ms with ED" test and the "50ms without ED" test to have a position difference of ≈ 9 yards.



(a) Without ED and 50ms latency



(b) With ED and 100ms latency

Figure 5.6: The illustrations show a situation where a player hits another player with an ability. P_1 is the target of the ability-hit event while P_2 makes the event. The illustrations contain four terms that must be defined: "Pos" is the position of a specific player, "Hit" is an ability-hit event created by a specific player, "A_t" means the event should be activated at time t and "Q_t" means the ability must be queued and should be activated at time t.
In both situations, P_2 sees the position of P_1 at time t_0 when the ability-hit event is activated. P_1 however, sees his position at time $t_0 + 100$ms when the ability-hit event is activated at P_1.

Agreement Test						
	With ED				Without ED	
Latency	25ms	50ms	100ms	250ms	25ms	50ms
Iterations	1423	1560	309	223	282	228
Number of hits	475	517	100	77	99	79
Agree on hit	474	518	85	10	93	64
Agree on no hit	946	1036	193	88	173	143
Disagreements	3 (0.21%)	6 (0.38%)	31 (10%)	125 (56%)	16 (5.7%)	21 (9.2%)
Agreements in %	99.79	99.62	90	44	94.3	90.8

Table 5.2: Test results for the agreement test

Test Results

The test results from the agreement and position prediction test cases are shown in Table 5.2 and Table 5.3 respectively.

The test results for the *event execution* test are shown in Figure 5.8 for cases with event delay and in Figure 5.7 for cases without event delay. The figures show absolute event execution time difference between peers.

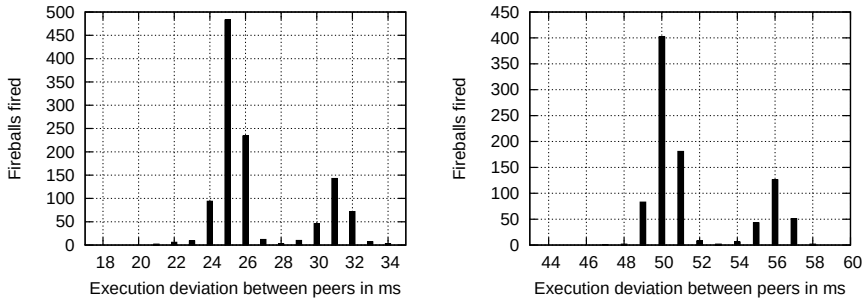
Agreement Test Table 5.2 shows the test results of the *agreement* test case. The results show that the test cases within our requirements pass the *agreement* test while the tests outside our requirements failed. The tests within our requirements are above the set limit of 99%. The best result of the test cases outside our requirements is the "25 ms without event delay" test where the actors agree in 94.3% of the iterations.

It is interesting to note how closely the results of the "100ms with ED" and "50ms without ED" cases resemble each other. Agreement is at 90% and 90.2% respectively. This is because the implemented event delay in effect cuts off 50ms of the latency by delaying the game execution 50ms. Therefore, the tests "100ms with ED" and "50ms without ED" show equal results. This further supports that the event delay works as intended and that event delay works by adding a global delay, as described earlier.

Event Execution Test Figure 5.8(a) and Figure 5.8(b) show the 25ms and 50ms tests with event delay, and lies within our requirements. The charts of the tests within our requirements show that most of the ability-fire events are activated at both peers within a few milliseconds.

Figure 5.8(c) and Figure 5.8(d) shows the 100ms and 250ms tests with event delay. These tests are outside the requirements of what Rawrlocks is intended to handle. Therefore, there is a time deviation too big to be acceptable. The 100ms and 250ms latency tests with event delay fails the *event execution* test.

Figure 5.7 shows the results of the tests without event delay. Even though the latency is within our requirements for both tests, the time difference be-



(a) 25ms latency

(b) 50ms latency

Figure 5.7: Fireball ability-fire event activation time deviation across clients. All tests are done without event delay.

tween ability-fire event activation deviates more than is allowed per our satisfiability metrics. It is worth noting that the ability-fire event activation deviates by an amount of time approximately equal to the network latency between peers. This is expected when latency is not taken into consideration in the game.

There is a spike in all the charts in Figure 5.8 around 20-50ms later than the biggest spike. In the charts in Figure 5.7 this spike occurs around 6-8ms later. All the values are from the first fireball fired in an iteration and not any additional fireballs. We assume this is due to a programming error and therefore we do not consider these to be useful results.

The test results show that our event delay technique succeeds in executing events on all peers within a reasonable timespan. This holds so long as the latency between peers does not exceed the amount of time events are delayed by.

Position prediction test Table 5.3 shows that there is a noticeable difference between tests within our requirements. The "25ms with ED" test has one occurrence (0.21%) outside our satisfactory metrics while the average position difference is 0.47 yards. The "50ms with ED" test has 36 occurrences (6.9%) outside our satisfactory metrics. The average position difference is below our satisfiability requirements in both tests within our requirements. In the test cases outside our requirements, the players disagree on positions in every test run.

In the "50ms with ED" test the maximum position difference is 2.7 yards, a distance beyond the satisfying requirements. For reference, the avatar radius is 23 yards, thus 2.7 yards is $\approx 12\%$ of the avatar radius. The movement speed of an avatar is 90 yards/s, meaning that a distance of 2.7 yards is coverable in 30ms. Furthermore the average difference does not exceed the limit of 1 yard. These facts make us believe that the erroneous position predicitions do not have a big impact.

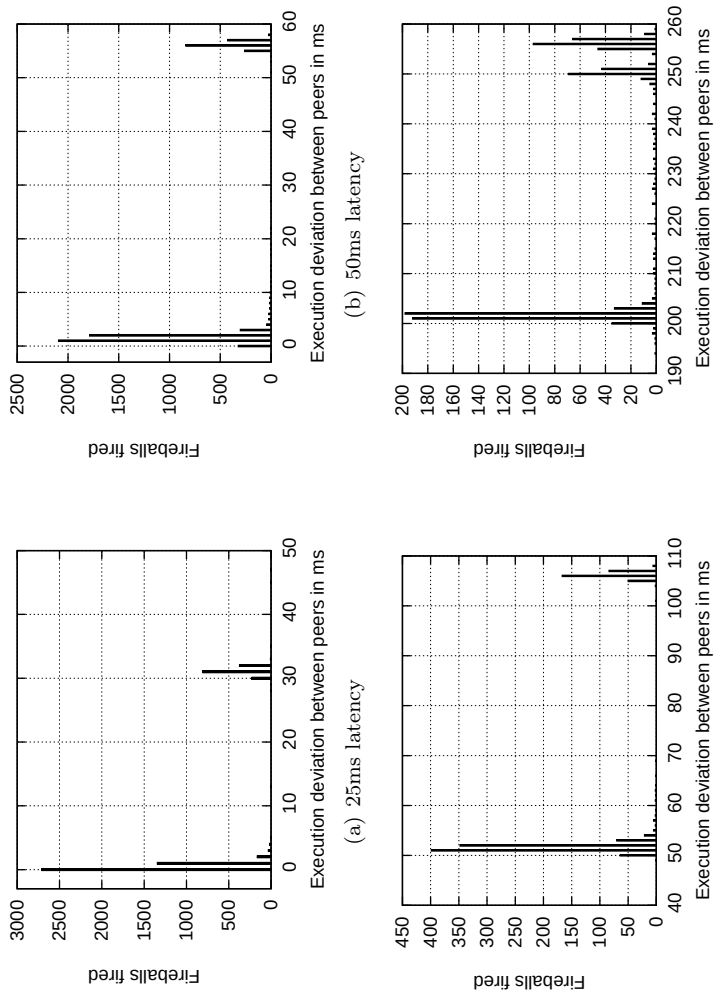


Figure 5.8: Fireball ability-fire event activation time deviation across clients. All tests are done with event delay.

Position Prediction Test						
	With ED				Without ED	
Latency	25ms	50ms	100ms	250ms	25ms	50ms
Iterations	474	518	100	77	99	79
Position difference (average) in yards	0.47	0.86	9.9	37	5.0	9.7
Position difference (max) in yards	1.7	2.7	13	50	5.1	24.4
Position difference above 1 yard	1	36	100	77	99	79
Position difference above 1 yard in %	0.21	6.9	100	100	100	100

Table 5.3: Test results for the position prediction test

Fireball Scenario Results						
	With ED				Without ED	
Latency	25ms	50ms	100ms	250ms	25ms	50ms
Agreement test	×	×				
Event execution test	×	×				
Position prediction test	×					

Table 5.4: The fireball scenario tests that satisfied our test metrics are marked with an "×".

Based on the results, we conclude that the functionality of the position prediction depends on the latency between peers. This is probably because we use extrapolation if the received position should be active at a time before the current game time. Therefore, in tests with 50ms delay, the position is set the moment it is received. After the position is received, we extrapolate between earlier received points.

Again there is a resemblance between the "100ms with ED" and "50ms without ED" tests. The average position difference is 9.9 yards and 9.7 yards respectively. We expected the positions to differ with ≈ 9 yards, so the results are as expected.

Satisfying Test Results:

Table 5.4 shows the tests that passed our requirements, illustrated by an "×". One test within our requirements failed, the *position prediction* "50ms with ED" test. Otherwise, the test results are as expected.

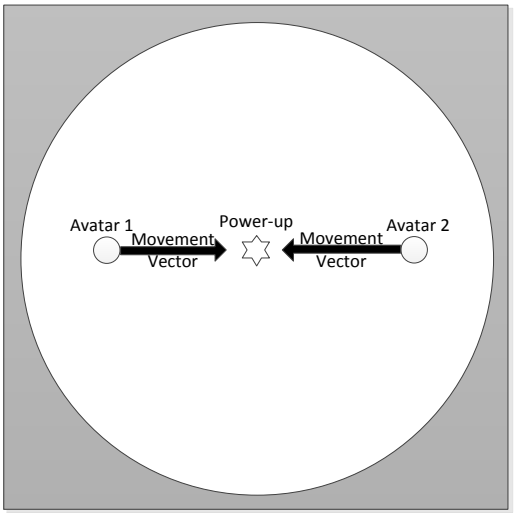


Figure 5.9: Illustration of Power-Up scenario. Two players race each other for the power-up.

5.2.2 Scenario 2: Power-Up

In this scenario two players race each other for a power-up by moving towards it from opposite sides. The ownership of the power-up belongs to player 2. The scenario is illustrated in Figure 5.9.

Variables We modify the latency in this scenario. First, the scenario is run with a latency of 25ms, 50ms, 100ms and 250ms with event delay enabled. Afterwards, the test is run with 25ms and 50ms latency without event delay.

Motivation This test is conducted to check two aspects of the SS implementation:

- Test to check if only one avatar obtains the power-up effect.
- Check if the owner of a power-up has an advantage in obtaining the power-up in relation to the position prediction. The implemented position prediction should cause the owner of the power-up to not have an advantage.

Measurements In this test we are interested in three metrics:

Agreement test We look at whether the players agree on who gets the power-up. The test results show the number of times the players agreed and disagreed.

Disagreement test If the players disagree on who gets the power-up, we look at collisions between players and power-up from the perspective of the player that did not get the power-up. The time difference between collisions is depicted in the test results.

Exactness test The power-up effect must only be active on one avatar at any time. The test results show the amount of times each player gets the power-up effect.

The tests satisfy our goals by the following metrics:

Agreement test The avatars start equally far away from the power-up and starts walking towards it at equal speeds at the same time. Therefore, the avatars should reach the power-up in the exact same frame. In order to meet the requirements, the players must agree on the outcome in at least 99% of the test runs.

Disagreement test When the players disagree on who gets the power-up, the time difference between the collisions between players and power-up must be below 10ms out of which 5ms stems from clock synchronisation inaccuracy. The value is so low to make sure it is unnoticeable for a human.

Exactness test The test satisfies iff one player gets the power-up. If more than one avatar gets the power-up, the test has failed. If the game is to remain consistent at latencies above GD, this test must not fail in any test case, with or without event delay.

Expected Outcome We expect the tests to satisfy our requirements if the latency is less than or equal to GD and event delay is activated. Otherwise, we expect the tests to fail. However, we expect the *exactness test* to succeed in every test case.

Test Results

The test results from the *agreement* test are shown in Table 5.5, the *disagree time difference* test results are shown in Table 5.6 and the *exactness* test results are shown in Table 5.7.

Agreement Test The results of the agreement tests are shown in Table 5.5. The results show that the players do not agree on who gets the power-up. The best result is for the "25ms with ED" test case where the players agree $\approx 66\%$ of the test runs. In the rest of the cases, the players agree less than 1% of the times. This is not satisfiable in itself because the players should agree on the result of ingame events.

The problem with this test is that the players start with an equal distance to the power-up and starts moving at the exact same time. Therefore, the smallest difference in player position can can result in a disagreement between players. The fact that the players agree $\approx 66\%$ of the times in the 25ms with event delay is impressive since the smallest incorrectness can affect the result.

It is interesting to look at the time difference between collisions with the power-up for each player in the eyes of the player that did not get the power-up. If the time difference is so small that humans are unable to notice it, the disagreement is irrelevant, since the player that disagreed with the event is unable to notice that the opponent had an unfair advantage.

Agreement Test						
	With ED				Without ED	
Latency	25ms	50ms	100ms	250ms	25ms	50ms
Iterations	3523	1362	329	245	270	212
Player 1 got power-up	1159	55	0	0	0	212
Player 2 got power-up	2366	1307	329	245	270	0
Agreement	2336	76	0	1	1	0
Disagreement	1177	1285	329	244	269	212

Table 5.5: Test results for the agreement test.

Disagreement Test The results for the *disagreement* test are shown in Table 5.6. The discussion of these results are split in two parts: The results from tests within- and outside our requirements.

Disagree Time Difference Test						
	With ED				Without ED	
Latency	25ms	50ms	100ms	250ms	25ms	50ms
Disagreement iterations	1177	1285	329	244	269	212
Disagreement time in ms (average)	0.35	0.67	50	200	24	49
Disagreement time in ms (maximum)	35	17	52	202	27	52
Over disagreement time-limit	9	16	329	244	269	212

Table 5.6: Test results for the disagree time difference test.

Inside requirements The tests in this discussion are 25ms and 50ms tests with event delay. In the 25ms test, the average amount of time between players colliding with the power-up is 0.35ms. A time frame so small is only possible if a game is running at more than 2500FPS. If the game ran with 20FPS, the logged time difference would be in iterations of 1/20th of a second. 1/20th of a second is 50ms, so every value below 50ms would be logged as a difference of 0ms. In both tests inside our requirements, the maximum disagreement time is 35ms. A number that is in the range from 0ms to 50ms, and thus can be in the same frame in a 20FPS scenario.

The number of frames required for the 50ms test is 1400, which is also well above a normal frame rate for games. The maximum amount of time between players colliding with the power-up is 17ms. This is also a value below what is humanly noticeable.

Outside requirements None of the tests outside our requirements satisfied our testing metric. Even though the differences can be unnoticeable by a human with 20FPS as described above, the disagreement average is too high compared to the *inside requirements* results where the average is below 1ms.

Exactness Test Table 5.7 shows the results of this test. The results show that exactly one player gets the power-up in every iteration and therefore satisfies the requirements completely. The fact that this test never fails shows that variables with high exactness requirements are exact even at high latencies. This is necessary if the game state is to remain consistent for peers where latency may exceed the specified maximum value.

Exactness Test						
	With ED				Without ED	
Latency	25ms	50ms	100ms	250ms	25ms	50ms
Iterations	3523	1362	329	245	270	212
Exact power-up pickup	yes	yes	yes	yes	yes	yes

Table 5.7: Test results for the exactness test.

Satisfying Test Results:

Table 5.8 shows the tests that passed our requirements, illustrated by an "×".

Power-Up Scenario Results						
	With ED				Without ED	
Latency	25ms	50ms	100ms	250ms	25ms	50ms
Agreement test						
Disagreement test	×	×				
Exactness test	×	×	×	×	×	×

Table 5.8: The test runs that had satisfactory results as per our requirements.

5.3 Bandwidth Testing

The test described in this section helps to determine if Rawrlocks abides by our bandwidth requirements of 256/256 kbit/s.

Scenario The main focus in this test is to determine the average bandwidth consumption for different numbers of players in the game. Certain events, such as firing abilities or picking up power-ups causes temporarily increase in the

Bandwidth Consumption Results			
Players	2	3	5
Connections for every peer	1	2	4
Runtime in seconds	206	257	150
Bandwidth consumption (avg) in kbit/s	11.1	21.7	43.1
Bandwidth consumption (max) in kbit/s	16.3	34.1	68.8
Bandwidth consumption per conn. (avg) in kbit/s	11.1	10.9	10.8
Bandwidth consumption per conn. (max) in kbit/s	16.3	17.1	17.2

Table 5.9: Results from the bandwidth consumption test

bandwidth consumption, so we have constructed a simple bot that activates these events.

Bot Functionality Without going into too much detail, we briefly describe the functionality of the bot.
The bot has two desires:

1. It wishes to stay away from the lava. If it is in the lava it uses teleport to get away.
2. It wishes to cause damage to its opponents. It uses scourge if another avatar is within a certain distance. Otherwise, it either decides to walk in a random direction or shoot a fireball in the direction of an opposing avatar.

Variables We change one value in this test: the number of players.
By changing the number of players, we get an estimate of the traffic increase that more peers cause. A fully connected topology implies that the number of messages sent is $(n-1) * PPS$ – where n is the number of peers in the network and PPS is the amount of packets per second. In Rawrlocks, minimum is $PPS = 20$.

Measurements We perform this test with two, three and five peers. This means that each peer connects to one, two and four peers respectively.

Expected Outcome We expect the tests to show that Rawrlocks, even with 10 players and at peak values, do not produce more traffic than can be sent and received with a 256/256kbit/s connection. We expect the bandwidth consumption to depend linearly on the amount of peers.

Test Results

Table 5.9 shows the results. The information in the table is made from outgoing traffic, but the incoming traffic is equal to the outgoing in Rawrlocks.

The results show that the outgoing bandwidth consumption per peer is approximately the same. With this in mind, the average outgoing bandwidth

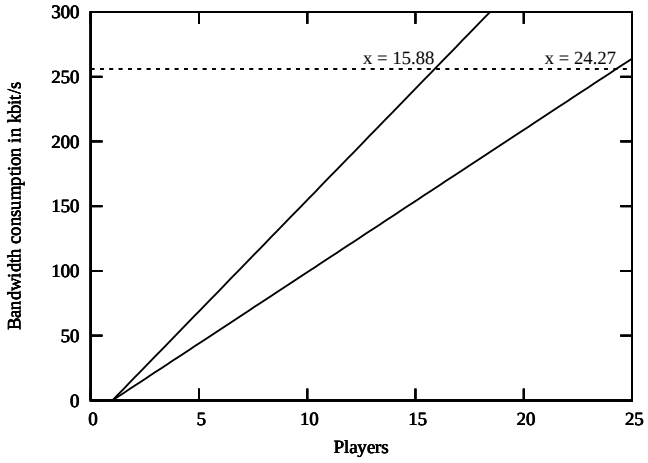


Figure 5.10: The required bandwidth as a function of the number of players in the game. If the players must be able to handle peak bandwidth consumption the game scales to 15 players, while it scales to 24 players if players must only handle average bandwidth consumption

consumption in a game with 10 players is approximately $(10-1) * 11\text{kbit/s} = 99\text{kbit/s}$.
The maximum outgoing bandwidth consumption in a game with 10 players is approximately $(10-1) * 17.2\text{kbit/s} = 154.8\text{kbit/s}$. A maximum bandwidth consumption value of 154.8 kbit/s is below our bandwidth limit of 256/256 kbit/s.

Figure 5.10 shows the bandwidth consumption as a function of the amount of players in the game. According to the graph, Rawrlocks scales to 15 players under maximum bandwidth consumption and 24 players with average bandwidth consumption.

5.4 Summary

The tests performed in this chapter are related to the problems outlined in the problem definition in Section 1.1. The overall problem is, "is it possible to keep a fast-paced multiplayer game with up to 10 players consistent using a peer-to-peer solution". We tested for consistency in Section 5.2 and bandwidth consumption in Section 5.3.

Consistency testing tests to see if the game remains consistent at variable latencies with event delay both enabled and disabled. We test two scenariors; a fireball scenario, described in Section 5.2.1 and a power-up scenario, described in Section 5.2.2. Table 5.1 shows the conducted tests within each scenario and which test cases are inside or outside our requirements.

The results from the fireball scenario are summed up in Table 5.4. Only one test within our requirements fail, the *position prediction* at 50ms latency

with event delay. The average results from the failed test case are within our satisfactory limit, but in 36/518 occurrences the position varies by more than one yard across nodes. The maximum variance is at 2.7 yards. The results show that the game fulfills the original requirements in all but one test case. In the failed test case, the results are arguably pretty good – the average position variance is less than one yard and the maximum variance 2.7 yards, a distance coverable in 30ms.

The results from the power-up scenario are summed up in Table 5.8. The results show that there are problems with agreeing on the outcome of events across peers. However, if the players disagree on the outcome, the difference in time between collisions is so small that it is impossible for humans to see the difference. The situation presented in the power-up scenario would rarely occur in a real world scenario. It is unlikely that players starts moving towards a power-up from an equal distance at the exact same time. Conflicts happen because the time frame between collisions is less than 1ms. The exactness test succeeds in all test cases and therefore the game does not reach an inconsistent state even at high latencies.

Bandwidth testing tests how much traffic Rawrlocks generates in a normal game. The results, shown in Table 5.9, show that the bandwidth consumption is below the limit of 256kbit/s, even during peaks. The bandwidth consumption per connection is almost the same at different values of players in the game. If we assume the bandwidth consumption per connection is constant we can calculate the theoretic maximum amount of players in a game of Rawrlocks. The result is shown as a graph in Figure 5.10. The graph shows that bandwidth consumption scales linearly with the number of players, and that Rawrlocks should be playable with 15 players at peak consumption.

6 Conclusion

In this chapter, we look at what has been done and how to proceed. The chapter also discusses the generality of the design of the synchronisation service and the event delay.

6.1 Conclusion

This report contains analysis of the problems related to a peer-to-peer network architecture in a fast-paced action game with a maximum of 10 players. We created a small test game in this genre called Rawrlocks. We focused on techniques to keep the game state consistent across peers. After analysing the variables in Rawrlocks we found found that each variable had different requirements on exactness and freshness. The variables with high exactness requirements and those with high freshness requirements had to be distributed in different ways.

For the variables with high freshness requirements, we proposed four solutions on how to keep the game state consistent across peers. The design of the chosen solution is described in Section 3.6.2. The solution uses exact updates and seeks to execute events at the same absolute time across peers. The solution provides a technique to predict the position of the local avatar such that the avatar's position is known when a remote player receives it.

For the variables with high exactness requirements we designed a synchronisation service, which is described in Section 3.4.2. The SS keeps variables exact by allowing only one peer, the owner of the variable, to commit changes to a specific variable. If other peers wish to alter a shared variable they must contact the owner of the variable first.

We conducted a number of tests on Rawrlocks to show that it meets the requirements specified in Section 1.1. The tests are split up in two parts: Consistency testing and bandwidth testing.
We performed two test scenarios in order to test the consistency of Rawrlocks: A fireball test, described in Section 5.2.1, and a power-up test, described in Section 5.2.2.

The fireball test fails in one test case within the requirements. There are problems regarding the position prediction in the test case with event delay and 50ms latency. However, we argue that the differences in player position

across clients in the tests does not have a major impact in a real world setting. The main reason for this is that the average position difference is below our limit of one yard.

The test results for the *event execution* test in the fireball scenario shows that the peers execute events within 3ms in absolute time in the 25ms and 50ms latency with event delay test cases.

The power-up test shows that if two players request a shared variable within a time frame of less than 1ms, the players are likely to disagree on who should be allowed access to the shared variable. We argue that this is not a problem since, the players request the object in a time frame so small, it is impossible for a human to see who should rightfully be granted access to the shared variable.

The exactness tests of the power-up scenario all succeed. This suggests that the game state remains consistent across peers even at latencies not intentionally supported by the solution.

The bandwidth test shows that the bandwidth consumption by Rawrlocks is below the bandwidth requirements of 256/256kbit/s even at peak values with the maximum amount of players in the game.

According to our test results, Rawrlocks is playable with up to 25ms latency. Furthermore, Rawrlocks should remain playable at 50ms latency. Even latencies above 50ms does not result in an inconsistent game state. Instead variables with high freshness requirements differs more and more between peers as latency increase.

6.2 Discussion

This section discusses the generality of our implementation. Specifically we discuss the generality of the SS and the event delay implementation.

6.2.1 Generality of Implementation

Generality is the ability to map our solutions onto other problems and games. E.g. how easy is it to refit our solution to a first person shooter game.

In order to make the solution presented in the report more general, events and the necessary event information must be defined in a general manner. No matter how general the definition of events is, the amount of data that must be transferred depends on the amount of events and the complexity of the events. This means, that more events means requires events must be specified and distributed between GCs. Thus, the bandwidth requirement increases for the game.

The SS is in itself general and can be used for other games without altering any code in the SS. This is because the SS is a variable store that can link up with other variable stores and keep the variables consistent across SSs. The SS still have limits by only allowing integers. This prevents using the SS for other tasks like a chat or floating point numbers. It is currently not possible to extend a text string for a chatlog or adding floating points.

The game implementation can be compared with the techniques used by the Source engine described in Section 2.7. The lag compensation that Source uses can be compared to our event delay in the sense that latency is taken into consideration to have a correct evaluation of the game. Prediction deployed by the Source engine can be compared to Rawrlocks position prediction. Every client is also a server and therefore try to predict how the other players move around.

Generality of Event Delay

The events in Rawrlocks have different necessary information that must be shared with the other peers. The amount of data that is sent to other peers depends on how many events the game contains and how much information each event contains.

The GC uses an *event delay*, defined in Section 4.3, in order to delay events and activate. The GC distributes the information of events via a custom made packet system. The packet system contains the necessary information needed to activate events that are distributed via the GC. In order to add new events to the GC, the necessary event information must be defined. Furthermore, an event has some base values and functions that must be defined. That is, which *conditions* must be true for the event to activate, what data is put in the *event queue*, how is the data added to the local packet and how to *activate* the event. Once an event contains this information, it can be added to the game.

The information required to create new events is game specific and not general. However, the overall design of events is general and can be used for any game that makes use of event delay.

6.2.2 Decentralisation and Latency

Latency affects a peer-to-peer game more than a client-server game because there is no central authority to enforce consistency. In the current implementation a single player with high latency to every other peer is likely to ruin the game experience for everybody. In a client-server environment, a peer with high latency to the server only ruins the game for himself because the server can force actions of clients.

Latency problems in a client-server environment merely cause problems for the clients and not the consistency of the game. Latency problems in a peer-to-peer environment is likely to affect consistency because a peer with high latency to other peers owns part of the game state.

6.3 Future Works

There are several directions to take from here, to improve Rawrlocks. Some of the ones think are interesting are listed in this Section.

6.3.1 More testing

There is always room for more testing. Currently, there are elements in the game that is not tested. These elements may contain flaws in the game. The elements that can be tested are as following:

Packet loss how does the game act when packets are lost.

45ms Latency test with Event Delay The problems with the 50ms latency tests may relate to the processing time and a hard limit of 50ms GD. A test with lower than 50ms can show if this assumption is true.

Latency hiding methods Implement more latency hiding methods and test them against each other.

Modifiable by all does variables that are modifiable by all stay consistent.

6.3.2 Interpolation

Positions of remote players could be interpolated between received points. Interpolation is described in Section 2.7.2. In Rawrlocks, we would interpolate between positions if we know the position of a remote player at a time $t_1 > t$ where t is the current game time. The remote player should interpolate between its current position and the known position. An interpolation implementation would likely reduce the effects incorrect position predictions.

6.3.3 NAT Handling

NAT is one of the places *Demigod* failed. We would implement and test a reliable way to handle NAT and keep the game playable.

6.3.4 Cheaters

Currently, the protocol allows cheater to do almost anything they would want to do.This is an obvious place to improve the implementation. A multiplayer game is not worth much in a real world scenario where the players are performs malicious deeds.

6.3.5 Object Migration

If a player disconnects, all the objects the player owns are lost. Migrating these in a meaningful fashion can improve the game as it is not destroyed when a player disconnects.

6.3.6 Apply to Another Game

The methods described in this report could be interesting to apply to another game to see exactly how general they are. There are several open source games

with multiplayer support, e.g. Quake 3, an FPS game with an open source engine [27]. Quake 3 has also been used as a test bed by others in *Donnybrook* [2].

By using the Quake 3 engine, new problems arise as the game is in 3D instead of 2D like Rawrlocks. Quake 3 is a more complex game than Rawrlocks, so a new set of events and a new packet structure is required.

If the model is applied and tested on a commercially succesful game, it would prove that the model works outside games designed for the purpose.

Bibliography

[1] Björn Knutsson, Massively Multiplayer Games, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games, 2004.

[2] Ashwin Bharambe, John Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *SIGCOMM '08: Proceedings of the 2008 conference on Applications, technologies, architectures, and protocols for computer communications*, August 2008.

[3] Blizzard Entertainment. Warcraft iii. `http://us.blizzard.com/en-us/games/war3/`.

[4] Gas Powered Games. Demigod. `http://www.demigodthegame.com/`.

[5] Brad Wardell CEO at Stardock. Demigod: So what the hell happened? `http://frogboy.impulsedriven.net/article/352561/Demigod_So_what_the_hell_happened`.

[6] IceFrog. Defense of the ancients. `http://www.playdota.com`.

[7] Stunlock Studios. Bloodline champions. `http://www.bloodchampions.com`.

[8] S2Games. Heroes of newerth. `http://www.heroesofnewerth.com`.

[9] Glenn Fiedler. Glenn fiedler's game development articles and tutorials. GDC 2010 Networked Physics Slides.

[10] Janus Hansen, Martin B. Rosenbeck, and Rune K. Jensen. Peer-to-peer middleware for fast-paced computer games, 2010.

[11] Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[12] Andrew Sears Julie A. Jacko. *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*. 2002.

[13] Harri Hakonen Jouni Smed, Timo Kaukoranta. Aspects of networking in multiplayer computer games, 2002.

[14] Kajal Claypool Mark Claypool. Latency and player actions in online games. 2006. `http://web.cs.wpi.edu/~claypool/papers/precision-deadline/`.

[15] Jin-Hee Choi and Chuck Yoo. One-way delay estimation and its application. *Computer Communications*, 28(7):819 – 828, 2005.

[16] Zymoran. Warlock brawl. `http://www.warlockbrawl.com/`.

[17] Gregor Maier, Fabian Schneider, and Anja Feldmann. NAT usage in residential broadband networks. In Neil Spring and George Riley, editors, *PAM '11: Proceedings of the 12th International Conference on Passive and Active Network Measurement*, volume 6579 of *Lecture Notes in Computer Science*, pages 32–41. Springer Berlin / Heidelberg, March 2011.

[18] Daryl Seah, Wai Kay Leong, Qingwei Yang, Ben Leong, and Ali Razeen. Peer nat proxies for peer-to-peer games. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, NetGames '09, pages 6:1–6:6, Piscataway, NJ, USA, 2009. IEEE Press.

[19] Marc Liberatore Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions on Networking*, 22:1–17, 2007.

[20] Virginia Lo Chris GauthierDickey, Daniel Zappala and James Marr. Low-latency and cheat-proof event ordering for peer-to-peer games, 2004.

[21] Source multiplayer networking. `http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`.

[22] Inc. BitTorrent. Bittorrent - delivering world's content. `http://www.bittorrent.com`.

[23] The OGRE Team. A 3d library for multiple desktop and mobile platforms. `http://www.ogre3d.org/`.

[24] J. Postel. RFC 768: User datagram protocol, August 1980. Status: STANDARD. See also STD0006.

[25] Song Ho Ahn. High resolution timer. `http://www.songho.ca/misc/timer/timer.html`.

[26] Linux Foundation. netem. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[27] iquake3 team. iquake3. `http://ioquake3.org/`.