UNIVERSITY OF GOTHENBURG

A Reliable Generic Game Server

Niklas Landin
Richard Pannek
Mattias Pettersson
Jonatan Pålsson

Abstract

This is the abstract!

Table of Contents

1 Introduction

Online gaming, and computer gaming in general has become an important part in many peoples day-to day lives. A few years ago, computer games were not at all as popular as they are today. With the advances in computer graphics and computer hardware today's games are much more sophisticated then they were in the days of *NetHack*, *Zork,* or *Pacman.*

The early computer games featured simple, or no graphics at all NetHack [2011]. The games often took place in a textual world, leaving the task of picturing the world up to the player. Multiplayer games were not as common as they are today, whereas most games today are expected to have a multiplayer mode, most early games did not.

Since these early games, the gaming industry have become much more influential in many ways. Many advanced in computer hardware are thought to come from pressure from the computer game industry. More powerful games require more powerful, and more easily available hardware. Due to the high entertainment value of modern computer games, gaming has become a huge industry, where large amounts of money are invested. The gaming industry is today, in some places even larger than the motion picture industry. Association [2011], Nash Information Services [2011]

Due to the increasing importance of computer gaming, more focus should be spent on improving the quality of the gaming service. As more and more computer games are gaining multiplayer capabilities, the demands for multiplayer networking software rises. This thesis is about techniques for improving the quality of this networking software.

The Reliable Generic Game Server, hereafter known as the GGS, is a computer program designed to *host* network games on one or more server computers. Hosting, in a network software setting, means allowing client software connect to the server software, for the purpose of utilizing services provided by the server. The GGS software provides games as a service, and the clients connecting to the GGS can play these games on the GGS.

The idea of game servers is not new, network games have been played for decades. Early, popular examples of network games include the *Quake* series, or the *Doom* games. Newer examples of network games include *World of Warcraft*, and *Counter-Strike*. The difference between the GGS and the servers for these games is that the servers for Doom, Quake, and the others listed, were designed with these specific games in mind.

What GGS does is to provide a *generic* framework for developing network games. The framework is generic in the sense that it is not bound to a specific game. There are many different types of games, some are inherently more time sensitive than others, strategy games are examples of games which are not very sensitive to time delays, first-person shooters however, can be very sensitive.

The generic nature of the GGS allows the creation of many different types of games, the motivation behind this is to remove the necessity of writing new game servers when developing new games.

The GGS is in addition to being generic, also *reliable* in the sense that the gaming service provided is consistent and available. A consistent and available server is a server that handles

hardware failures and software failures gracefully. In the event of a component breaking within the GGS, the error is handled by fault recovery processes, thereby creating a more reliable system.

1.1 Background

The game industry is a quickly growing industry with high revenues and many clever computer scientists. Strangely enough their customers often experience long downtimes due to maintaining or because of problems with the servers Terdiman [2006]. This is a problem that has existed and been resolved in other industries. The telecom industry, for instance, has already found solutions to similar problems.

A common figure often used in telecoms is that of *the nine nines*, referring to 99.999999999% of availability Armstrong [2003], or roughly $15ms$ downtime in a year. The level of instability and bad fault tolerance seen in the game server industry would not have been accepted in the telecom industry. This level of instability should not be accepted in the game server industry either. An unavailable phone system could potentially have life threatening consequences, leaving the public unable to contact emergency services. The same cannot be said about an unavailable game server. The statement that game servers are less important than phone systems are not a reason not to draw wisdom from what the telecoms have already learned.

Moving back to the gaming industry. The main reason to develop reliable servers is a higher revenue, do archive this it is important for game companies to expand their customer base. Reliable game servers will create a good image of the company. In general the downtime of game servers is much higher than the downtime of telecom systems evenso the overall structure of the systems is similar in many ways. It should be possible to learn and reuse solutions from the telecom systems to improve game servers.

In the current state game servers are developed on a per-game basis, often this seems like a bad solution. Developers of multiplayer games need to understand network programming, which above all be a problem for small companies and indipendent game developers who often lack expertise in that field. A way to help them in the competition would be to offer a generic game server which gives them a environment in which they can implement their game in. This approach would not only make it easier to develop network games, it would also allow games in different programming languages to be implemented using the same server.

Some key factors to the development of the GGS have been isolated. Many of these are found in the telecom sector too. The factors are *scalability, fault tolerance* and a *generic* nature. These terms are defined below.

Scalability (see 2.7) in computer science is a large topic and is commonly divided into subfields, two of which are *structural scalability* and *load scalability* Bondi [2000]. These two issues are addressed in this thesis. Structural scalability means expanding an architecture, e.g. adding nodes to a system without requiring modification of the system. Load scalability means using the available resources in a way which allows handling increasing load, e.g more users, gracefully.

Fault tolerance (see 2.5) is used to raise the level of *dependability* in a system, so that the dependability is high even in presence of errors. Dependability is the statistical probability of the system functioning as intended at a given point in time. Fault tolerance is the property of a system always to follow a specification, even in the presence of errors. The specification could

define error handling procedures which activate when an error occurs. This means that a fault tolerant, dependable system, will have a very high probability of functioning at a given point in time, and is exactly what is desired. Gärtner [1999]

A generic (see 2.4) game server has to be able to run different client-server network games regardless of the platform the clients are running on. It runs network games of different type. A very rough separation of games is real time games and turn based games.

The server behaves in a way similar to an application server, but is designed to help running games. An application server provides processing ability and time, therefore it is different from a file- or print-server, which only serves resources to the clients.

The most common type of application servers are web servers, where you run a web application within the server. The application server provides an environment and interfaces to the outer world, in which applications run. Hooks and helpers are provided to use the resources of the server. Some examples for web application servers are the *Glassfish* server which allows running applications written in Java or the *Google App Engine* where you can run applications written in Python or some language which runs in the *Java Virtual Machine*. An example of an application server not powering web applications, but instead regular business logic, is Oracle's *TUXEDO* application server, which can be used to run applications written in COBOL, C++ and other languages.

A database server can also be seen as an application server. Scripts, for example SQL queries or JavaScript, are sent to the server, which runs them and returns the evaluated data to the clients.

One purpose of this thesis is to investigate how one could make a game server as generic as possible. Some important helpers are discussed, such as abstraction of the network layer, data store and game specific features.

A prototype has been developed in order to aid the discussion of the theoretical parts of the GGS. The prototype does not feature all the characteristics described in this thesis. A selection has been made among the features and the most important ones have been implemented either full or in part in the prototype.

The choice of the implementation language for the prototype of the GGS was made with inspiration from the telecom industry. The Erlang language was developed by the swedish telecom company Ericsson to develop highly available and dependable telecom switches. One of the most reliable systems ever developed by Ericsson, the AXD301 was developed using Erlang. The AXD301 has possibly the largest code base even written in a functional language [Armstrong, 2003]. The same language is used to develop the prototype of the GGS. The usage of Erlang in the GGS is discussed in further detail in section 3.2.

1.2 Purpose

The purpose of creating a generic and fault tolerant game server is to provide a good framework for the development of many different types of games. Allowing the system to scale up and down is a powerful way to maximize the usage of physical resources. By scaling up to new machines when load increases, and scaling down from machines when load decreases costs and energy consumption can be optimized.

Fault tolerance is important for the GGS to create a reliable service. The purpose of a reliable game server is to provide a consistent service to people using the server. Going back to the telecom

example, the purpose of creating a reliable telecom system is to allow calls, possibly emergency calls, at any time. Should the telecom network be unavailable at any time, emergency services may become unavailable, furthermore the consumer's image of the telecom system degrades.

Returning to the game industry, emergency services will not be contacted using a game server, however an unavailable server will degrade the consumer's image of the system. Consider an online casino company. The online casino company's servers must be available at all times to allow customers to play. If the servers are unavailable customers cannot play and the company loses money. In this scenario an unavailable server can be compared to a closed real-world casino.

1.3 Challenges in developing the prototype

The word *generic* in the name of the GGS implies that the system is able to run a very broad range of different code, for instance code written in different programming languages or code written for a broad range of different game types. To support this, a virtual machine (VM) for each *game development language* (hereafter GDL for brevity) is used.

No hard limit has been set on which languages can be used for game development on the GGS, but there are several factors which should be taken into consideration when deciding the feasibility of a language:

- How well it integrates with Erlang, which is used in the core the GGS system?
- How easy it is to send messages from the GGS to the GDL VM?
- How easy it is to send messages from the GDL VM to the GGS?
- Is it possible to sandbox every game with a context or something comperable?

Internally the GDL VM needs to interface with the GGS to make use of the helpers and tools that the GGS provides. Thus an internal API has to be designed to make the GDL VM to be able to interacti with the GGS. This API is ideally completely independent of the GDL, and reusable for any GDL.

The communication with the gaming clients has to take place with help a protocol. Ideally a standard protocol should be used in order to shorten the learning curve for developers and also make the system as a whole less obscure. A major challenge during this project is to decide whether an existing protocol can be used, and if not, how a new protocol can be designed which performs technically as desired, while still being familiar enough to existing developers.

A great deal of work is devoted to make the GGS *reliable*. This includes ensuring that the system scales well and to make sure it is fault tolerant. In order to facilitate scalability the GGS needs a storage platform which is accessible and consistent.

1.4 Limitations of the prototype

The implementation of the GGS protocol together with storage possibilities, server capacity, and game language support imposes some limitations on the project. To get a functional prototype some limits must be set on the types games that can be played on the prototype.

The UDP protocol is not supported for communication between client and server. The TCP protocol was chosen in favor of UDP, due to the fact that the implementation process using TCP

was faster and easier than if UDP would have been used. UDP is generally considered to be faster than TCP for the transfer of game (and other) related data, this is discussed in more depth in 2.3 on page 8. In short, the decision of using TCP means that games that requires a high speed protocol will not be supported by the GGS prototype. Another limitation necessary to set on the system is the possibility to have huge game worlds due to the implementation of the scaling mechanism in the prototype.

In real time games all players are playing together at the same time. Latency is a huge problem in real time games, a typical round trip time for such games are one of 50 to 150ms and everything above 200ms is reported to be intolerable (see Färber [2002]). Latency sensitive games include most of the first person shooters with multiplayer ability, for example *Counter Strike* or massively multiplayer online role playing games (MMORPGs), for example *World of Warcraft*.

In turn based games each player has to wait for their turn. Latency is not a problem since the gameplay does not require fast interactions among the players, long round trip times will not be noticed. Examples of turn based games include board and card games, as well as multiplayer games like *Jeopardy*. Both game types have varying difficulties and needs when it comes to implementing them, a Generic Game Server should address all of them and help the developer to accomplish his goal.

Due to the limited capability of threading in many GDL VMs, the GGS prototype will not support MMORPGs.

The implementation of the GGS described in this thesis is only a small prototype and tests will be performed on simple games like pong or chess, thus there is no need to implement more advanced features in the system. Note that these limitations only apply for the prototype of the project, and that further developments to the GGS could be to implement these features.

1.5 Method

A prototype was developed early on in the project to carry out experiments. Using this prototype, the system was divided into modules. A demand specification was created, using this specification, the modules were easily identifiable.

The first prototype of the GGS consisted of simple modules, however, due to the separation of concerns among the modules, they were easily independently modified and improved. Once the basic structure of the GGS had been established, the first prototype was removed, remaining was the structure of the modules and the internal flow of the application. This could be seen as an iterative workflow, with the first prototype being the first iteration. The second iteration later became the final result of the GGS.

The layout of the GGS is both layered and modular. The first layer handles the most primitive data and produces a higher level representation of the data, passing it along to different modules of the GGS. The modular structure of the GGS plays an important role in making the system fault tolerant. The approach to fault tolerance is by replication, and restarting the faulting modules with the last known good data.

An informal specification and list of requirements of the system was outlined early on in the project. Usability goals for developers were set. During the project several demo applications were constructed, by constructing these applications, the usability goals were enforced.

2 Theory behind the GGS

In this chapter, the theory behind the techniques used in the GGS are discussed. Performance issues and the measuring of performance is discussed. Benchmarking techniques are discussed. The options when choosing network protocols are given, along with a discussion of each alternative. Finally, an overview of scalability, fault tolerance and availability are presented.

2.1 Design of the GGS system

The GGS is modeled after a real world system performing much of the same duties as the GGS. This is common practice [Armstrong, 2011] in the computer software world to understand complex problems more easily. While there may not always be a real world example of a system performing the exact duties of the system being modeled in the computer, it is often easier to create and analyze requirements for real world systems and processes than systems existing solely in virtual form in a computer. The requirements and limitations imposed on the real-world system can, using the proper tools, be transferred in to the software.

The real world system chosen for the GGS is a "Chess club" - a building where chess players can meet and play chess. Since a real-world scenario is readily available, and to such a large extent resembles the computer software required for the GGS, the next step in developing the GGS system is to duplicate this real world scenario in a software setting.

Some requirements, limitations and additions were made to the chess club system, so that the system would more easily and efficiently be replicated in a software setting.

In the text below, two examples will be presented. On example is that of a real-world chess club, in which players meet to play chess against each other, the other example is the GGS, and how it corresponds to this chess club. In figure 2.1 on the next page a graphical representation for the chess club is presented. The club is seen from above. The outermost box represents the building. In the GGS setting, the building would represent one instance of the GGS. Several buildings linked together would represent a cluster of GGS instances. In order for a player (the P symbol in the graphic) to enter the theoretical chess club, the player must pass by the entrance. By having each player pass by the entrance, a tally can be kept, ensuring that there are not too many players within the building. In the GGS setting, too many players entering would mean too many connections have been accepted by the GGS system, and that the structure of the system thus must be modified, adding additional servers.

Once a player has been allowed in to the chess club the player is greeted by the host of the chess club, in the GGS setting represented by the *Coordinator*, and is seated by a table. The coordinator keeps track of all the players in the building, and all moves made by the players. The information available to the coordinator means that cheating can be monitored and book keeping can be performed by this entity.

A player can only move the figures on her table in the chess club thus every game is isolated to a table, just as expected. This means that communication during a game only has to pass by the players of that particular game and the coordinator, making sure that no cheating takes place.

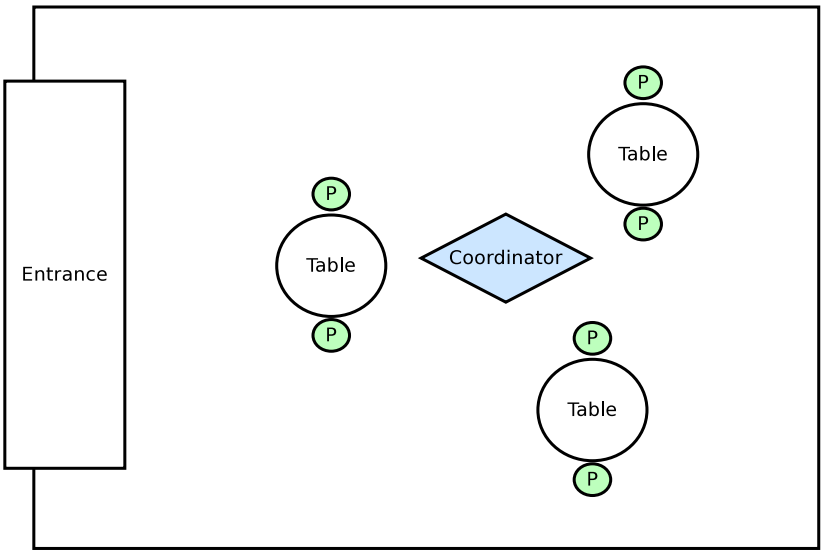

Figure 2.1: The layout of a physical "Chess club" with two players (P) sitting by each chess table (Table), a coordinator keeps track of all movements of players in the building. A player has to pass by the entrance to enter or exit the building. The building is represented by the outermost box.

This isolation of the games play an important part in many properties of the GGS, the isolation means that games can for example be transferred among different chess clubs. Furthermore, if cheating takes place, corruption can only occur in the particular table where it was found and cannot spread.

Moving chess players from one location to another is one alteration made to the real world chess club system to make the system more appropriate for a software setting. Allowing games to be transferred is not an attribute usually desired in a real world chess club, where transferring players would mean moving the players from one building to another. In the software setting, moving players means moving the game processes from one system to another, perhaps to balance the system load. This transfer of players can occur transparently, without notifying the players.

The simplified life cycle of a game in the GGS can be viewed using algorithm 2.1 on the following page. To make this life cycle as efficient and useful as possible the scalability, fault tolerance and generic traits are being added to the GGS. These are not shown in the algorithm because these traits are tools in making the algorithm behaves as efficient as possible and are not the main focus when studying the life cycle of a game.

The limits imposed in 2.1 on the next page are arbitrary for this example, there are for example no limits in the GGS on the number of players connecting.

Algorithm 2.1 A very simple example of the flow through the GGS system when a game is played.

```
1: while players < 2:
2:    if a player connects, call connected
3: while the game commences:
4:    call the function game
5: when the game has stopped
6:    call the function endGame
7: function connected:
8:    assign the new player an id
9:    alert the coordinator of the new player
10:   if a free table does not exist:
11:       the coordinator creates a new table
12:   the coordinator places the player by the table, and begins watching the player
13: function game:
14:   perform game-specific functions. In chess, the rules of chess are placed here
15: function endGame:
16:   alert the coordinator, unregistering the players
17:   disconnect the players from the system, freeing system resources
```

2.2 Performance

There are many ways in which performance could be measured. For the clients, time and response times are useful measurements in time critical settings. In non-time critical settings, the reliability of message delivery may be an even more important factor than speed.

In a first person shooter game, the speed of delivery of messages with information about the current positions of all players is essential. The failure to deliver messages in time results in choppy gameplay for the players. In strategy games, the reliability of delivery may be more important than the speed, since the game is not perceived as choppy even if the messages are delayed.

For someone operating a GGS, it is perhaps more interesting to measure the system load, memory consumption, energy consumption and network saturation. These topics are discussed in theory in this section. The practical results for the prototype is discussed in chapter 3 on page 16.

2.3 Choosing a network protocol

There are two main types of protocols with help of which computer communication over the Internet usually takes place; TCP and UDP which are known as the network layer protocols and HTTP which is the most prominent application layer protocol. The transport layer protocols are commonly used to transport application layer protocols such as HTTP. TCP and UDP cannot be used on their own without an application layer protocol on top of them. Application layer protocols such as HTTP on the other hand need a transport layer protocol in order to work.

2.3.1 UDP

Many online games use UDP as the carrier for their application layer protocol. UDP moves data across a network very quickly, however it does not ensure that the data transferred arrives in consistent manner. Data sent via UDP may be repeated, lost or out of order. To ensure that the data is transferred is in good shape, some sort of error checking mechanisms must be implemented. UDP is a good choice for applications where it is more important that data arrives in a timely

manner than that all data arrives undamaged, it is thus very suitable for media streaming for example.

2.3.2 TCP

For reliable transfers TCP is often used on the Internet. Built in to the protocol are the error checking and correction mechanisms missing in UDP. This ensures the consistency of data, but also makes the transfer slower than if UDP had been used.

2.3.3 HTTP

Since HTTP is so widely used in web servers on the Internet today, it is available on most Internet connected devices. This means that if HTTP is used in the GGS, firewalls will not be a problem, which is a great benefit. However, due to the intended usage of HTTP in web servers, the protocol was designed to be stateless and client-initiated. In order to maintain a state during a game session using HTTP, some sort of token would have to be passed between client and server at all times, much like how a web server works. These facts combined make HTTP inappropriate for use in the GGS, since the GGS requires a state to be maintained throughout a session and also needs to push data from the server to clients without the clients requesting data. It should also be mentioned that HTTP uses the TCP protocol for transport.

2.3.4 The GGS Protocol

HTTP was designed to be a stateless protocol, which by adding some overhead is able to remove the need of a permanent connection and a state for each client. The GGS however already has a permanent connection to each client because it needs to push information to the clients. Therefore it is able to use the state to minimize the overhead in the communication between server and client. Therefore it was decided to invent a new protocol which was human readable like HTTP but customized for this special use. The GGS protocol is described in more detail in section 3.3.3.

2.4 Generic structure of the GGS

The GGS is a game server. It was made with a desire to be suitable for any kind of game. Any game with a client-server behavior should be perfectly suited for the GGS. A game should not only be able to vary in terms of genre, graphics, gameplay etc, but also in the way the game is implemented for example in different programming languages. The GGS should be OS independent and run on Windows, OSX and Linux. The GGS can be run as a listen server on the players computer and host games locally. It could also be a dedicated server running on dedicated independent hardware. It is meant to run any game in any environment in any way desired, therefore being as generic as possible.

Another aspect was the desire to let a client upload the source code of the game it would like to play on the GGS. This way every client could connect to the server and install the game through a without the need of installation through the server provider or maintainer.

2.5 Fault tolerance

Fault tolerance is an important factor in all servers, a server that is fault tolerant should be able to follow a given specification when parts of the system fails. This means that fault tolerance

is different in each system depending on what specification they have. A system could be fault tolerant in different aspects, one is where the system is guaranteed to be available but not safe and it could also be reversed, that the system is safe but not guaranteed to be available. Depending on the system one property may be more important. A system could also have non existent fault tolerance or it could be both safe and guaranteed to be available. It should be noted that it is not possible to achieve complete fault tolerance, a system will always have a certain risk of failure. With this in mind the goal is to make the GGS prototype as fault tolerant as possible.

In order to make the GGS prototype fault tolerant the programming language Erlang has been used. Erlang will not guarantee a fault tolerant system but it has features that support and encourage the development of fault tolerant systems. In the GGS it is important that the system overall is fault tolerant and not small parts only. Crashes of the whole system should be avoided as this would make the system unusable for a time. By using supervisor structures it is possible to crash and restart small parts of the system, this is convenient as fault can be handled within small modules thus never forcing a crash of the system.

The need for fault tolerance in game servers is not as obvious as it may be for other type of servers. In general all servers strive to be fault tolerant as fault tolerance means more uptime and a safer system. This applies to game servers as well, good fault tolerance is a way of satisfying customers. In general, game servers differ from many other fault tolerant systems in that high-availability is more important than the safety of the system. For example a simple calculation error will not be critical for a game server but it may be in a life-critical system and then it is better that the system crashes than works with the faulty data. There are cases where safety may be critical in game servers, one example is in games where in-game money exist.

2.6 Availability

One important factor of any server is the availability. A server to which you are unable to connect to is an useless server. Other then within telecommunication, their uptime is of about 99,9999999%, the game developer community has not approached this problem very genuinely yet so there is much room for improvement.

There are several good papers on how to migrate whole virtual machines among nodes to replicate them but for the GGS a different approach has been chosen. Instead of duplicating a virtual machine, an attempt to lift the state of the VM to a storage external to the VM is made. The state is stored in a fast, fault tolerant data store instead of inside the VM. Some of them are *hot code replacement*, where code can be updated while the application is running and without the need to restart it, the *supervisor structure* provided by *OTP* and the inter node and process communication via *messages* instead of shared memory. We will discuss each of them later on.

2.7 Scalability

Each instance of the GGS contains several tables. Each table is an isolated instance of a game, for example a chess game or a poker game. The way that the GGS scales is to distribute these tables on different servers. In many games it is not necessary for a player to move among tables during games. This is for example not a common occurrence in chess, where it would be represented as a player standing up from her current table and sitting down at a new table, all within the same

game session. Therefore, the main focus of the GGS is not to move players among tables, but to keep a player in a table, and to start new tables instead. When a server has reached a certain number of players the performance will start to decrease. To avoid this the GGS will start new tables on another server, using this technique the players will be close to evenly distributed among the servers. It is important to investigate and find out how many players that are optimal for each server. This approach makes it possible to use all resources with moderate load, instead of having some resources with heavy load and some with almost no load.

As mentioned in the purpose section there are two different types of scalability, structural scalability and load scalability. To make the GGS scalable both types of scalability are needed. Structural scalability means in our case that it should be possible to add more servers to an existing cluster of servers. By adding more servers the limits of how many users a system can have is increased. Load scalability in contrast to structural scalability is not about how to increase the actual limits of the system. Instead it means how good the system handles increased load. The GGS should be able to scale well in both categories.

2.7.1 Load balancing

The need for load balancing varies among different kind of systems. Small systems that only use one or a couple of servers can cope with a simple implementation of it, while in large systems it is critical to have extensive and well working load balancing. The need also depends on what kind of server structure that the system works on. A static structure where the number of servers are predefined or a dynamic structure where the number varies.

Load balancing and scaling is difficult in different scenarios. When running in a separate server park, there are a set number of servers available, this means that an even distribution on all servers is preferable. When running the GGS in a cloud, such as Amazon EC2, it is possible to add an almost infinite number of servers as execution goes on. In this cloud setting, it may be more important to evenly distribute load on newly added servers.

Two methods of balancing load (increasing structure):

- Fill up the capacity of one server completely, and then move over to the next server
- Evenly distribute all clients to all servers from the beginning, when load becomes too high on all of them, then comes a new problem:
  - How do we distribute load on these new servers?

Load balancing is a key component to achieve scalability in network systems. The GGS is a good example of a system that needs to be scalable, to attain this load balancing is necessary. Optimization of the load balancing for a system is an important task to provide a stable and fast load balancer. There are certain persistence problems that can occur with load balancing, if a player moves from a server to another data loss may occur. This is an important aspect to consider when the load balancer is designed and implemented.

Load balancing can often be implemented using dedicated software, this means that in many applications load balancing may not be implemented because it already exist functional solutions. This depends on what specific needs the system have and a minor goal of the project is to analyze

Algorithm 2.2 A simple (insufficient) generator for identifiers

```
1: global variable state := 0
2: function unique
3:     state := state + 1
4:     return state
```

whether the GGS project can use existing load balancing tools or if it is necessary to implement load balancing in the project.

2.7.2 UUID

Inside the GGS, everything has a unique identifier. There are identifiers for players, tables and other resources. When players communicate amongst each other, or communicate with tables, they need to be able to uniquely identify all of these resources. Within one machine, this is mostly not a problem. A simple system with a counter can be imagined, where each request for a new ID increments the previous identifier and returns the new identifier based off the old one, see algorithm 2.2. This solution poses problems when dealing with concurrent and distributed systems. In concurrent systems, the simple solution in algorithm 2.2 may yield non-unique identifiers due to the lack of mutual exclusion.

The obvious solution to this problem is to ensure mutual exclusion by using some sort of lock, which may work well in many concurrent systems. In a distributed system, this lock, along with the state, would have to be distributed. If the lock is not distributed, no guarantee can be made that two nodes in the distributed system do not generate the same number. A different approach is to give each node the ability to generate Universally Unique Identifiers (UUID), where the state of one machine does not interfere with the state of another.

According to Leach and Salz [1998], "A UUID is 128 bits long, and if generated according to the one of the mechanisms in this document, is either guaranteed to be different from all other UUIDs/GUIDs generated until 3400 A.D. or extremely likely to be different". This is accomplished by gathering several different sources of information, such as: time, MAC addresses of network cards, and operating system data, such as percentage of memory in use, mouse cursor position and process IDs. The gathered data is then *hashed* using an algorithm such as SHA-1.

When using system wide unique identifiers, such as the ones generated by algorithm 2.2 with mutual exclusion, it is not possible to have identifier collisions when recovering from network splits between the GGS clusters. Consider figure 2.2, where *Site A* is separated from *Site B* by a faulty network (illustrated by the cloud and lightening bolt). When the decoupled node and the rest of the network later re-establish communication, they may have generated the same IDs if using algorithm 2.2, even when mutual system-wide exclusion is implemented. This is exactly the problem UUIDs solve.

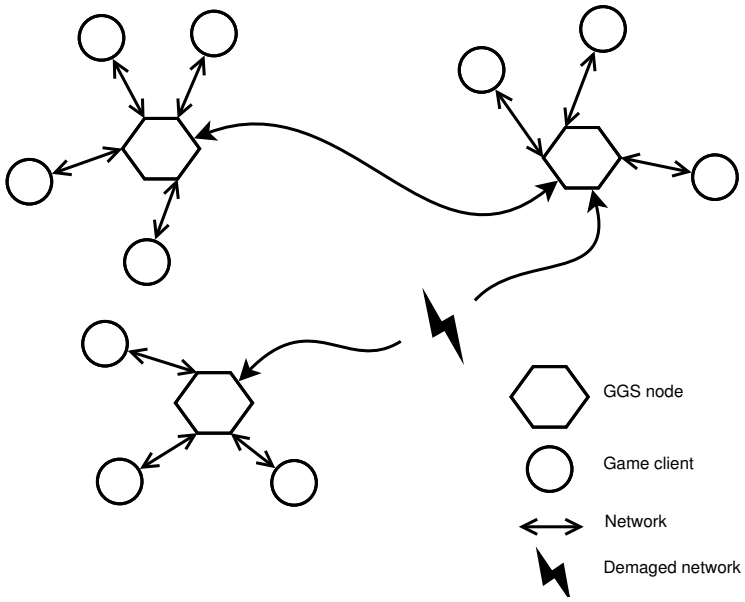

Figure 2.2: An example of a network split

2.8 Security

We only support languages running in a sandboxed environment. Each game session is started in its own sandbox. The sandboxing isolates the games in such a way that they cannot interfere with each other. If sandboxing was not in place, one game could potentially modify the contents of a different game. A similar approach is taken with the persistent storage we provide. In the storage each game has its own namespace, much like a table in a relational database. A game is not allowed to venture outside this namespace, and can because this not modify the persistent data of other games. of this not modify the persistent data of other games.

2.9 Game Development Language in a Virtual Machine

There is only a very limited number of game developers who would like to write their games in Erlang, therefore we had to come up with something to resolve this problem. The main idea was to offer a replaceable module which would introduce an interface to different virtual machines which would run the game code. This way a game developer can write the game in his favorite language while the server part still is written in Erlang and can benefit from all its advantages.

2.9.1 JavaScript

JavaScript has gained a lot of popularity lately, it is used in large projects such as *Riak*[1], *CouchDB*[2]. On the popular social coding site *GitHub.com*, 18%[3] of all code is written in JavaScript. The popularity of JavaScript in the programming community, in combination with the availability of several different JavaScript virtual machines was an important influence in choosing JavaScript as the main control language for our GGS prototype.

[1] `http://wiki.basho.com/An-Introduction-to-Riak.html`
[2] `http://couchdb.apache.org`
[3] during the writing of the thesis the percentage went up to 19% `https://github.com/languages/`

2.9.2 Other languages

Other languages like *lua*, *ActionScript* are suitable as well because there is a virtual machine for each of them which can be "plugged in" into our GDL VM interface. With help of the *Java Virtual Machine* or the *.NET* environment it is even possible to run nearly every available programming language in a sandbox as a GDL.

Due lack of time we have decided to use just the Erlang <-> JavaScrThere is only a very limited number of game developers who would like to write their games in Erlang, therefore we had to come up with something to resolve this problem. The main idea was to offer a replacable module which would introduce an interface to different virtual machines which would run the game code. This way a game developer can write the game in his favorite language while the server part still is written in Erlang and can benefit from all of its advantages.

2.9.3 JavaScript

JavaScript has gained a lot of popularity lately, it is used in large projects such as *Riak*[4], *CouchDB*[5]. On the popular social coding site *GitHub.com*, 18%[6] of all code is written in JavaScript. The popularity of JavaScript in the programming community, in combination with the availability of several different JavaScript virtual machines was an important influence in choosing JavaScript as the main control language for our GGS prototype.

2.9.4 Other languages

Other languages like *lua*, *ActionScript* are suitable as well because there is a virtual machine for each of them which can be "plugged in" into our GDL VM interface. With help of the *Java Virtual Machine* or the *.NET* environment it is even possible to run nearly every available programming language in a sandbox as a GDL.

Due lack of time we have decided to use just the Erlang <-> JavaScript bridge with our interface.

2.10 Testing

There are several ways in which the GGS can be tested. The most important aspect has been deemed to be the experience players have when using the GGS. In order to test the user experience of the GGS, a realistic usage scenario has to be set up.

The GGS is intended to be used for powering games which have many concurrent players. The players need not participate in the same instance of the game, games such as chess are prime candidates for the GGS.

When developing the GGS, two main categories of games exhibit in. There is only a very limited number of game developers who would like to write their games in Erlang, therefore we had to come up with something to resolve this problem. The main idea was to offer a replacable module which would introduce an interface to different virtual machines which would run the game code. This way a game developer can write the game in his favourite language while the server part still is written in Erlang and can benefit from all of its advantages.

[4] `http://wiki.basho.com/An-Introduction-to-Riak.html`
[5] `http://couchdb.apache.org`
[6] during the writing of the thesis the percentage went up to 19% `https://github.com/languages/`

2.10.1 JavaScript

JavaScript has gained a lot of popularity lately, it is used in large projects such as *Riak*[7], *CouchDB*[8]. On the popular social coding site *GitHub.com*, 18%[9] of all code is written in JavaScript. The popularity of JavaScript in the programming community, in combination with the availability of several different JavaScript virtual machines was an important influence in choosing JavaScript as the main control language for our GGS prototype.

2.10.2 Other languages

Other languages like *lua*, *ActionScript* are suitable as well because there is a virtual machine for each of them which can be "plugged in" into our GDL VM interface. With help of the *Java Virtual Machine* or the *.NET* environment it is even possible to run nearly every available programming language in a sandbox as a GDL.

Due lack of time we have decided to use just the Erlang <-> JavaScrg different performance demands were identified; real-time games and turn-based games. The real-time games were deemed more demanding than the turn based games. Tests were carried out using a real time game, since this is the more demanding type of games.

The real time game chosen for testing the GGS is *Pong*, a game in which two players play a game involving a all and two paddles. The goal for each player is to shoot beside the other players paddle while not allowing the ball to pass by her own paddle. The game requires real time updates and is quite demanding when played in several instances concurrently.

There has been some work on the area of testing game servers, see Lidholt [2002], who describes a test bench using *bots* for testing his generic hazard-gaming server. Lidholt describes how his server, capable of running several different casino games is tested using artificial players, so called bots. Performance is measured in "number of clients" able to connect to the server, and the system load.

Similar tests were performed on the GGS, and the results of these tests are visible in chapter 5. The tests were initially performed by starting an operating system process for each player. Due to lack of hardware, not enough player processes could be started in this way. The bots were re-written in Erlang, and due to Erlang's light weigh threads, enough processes could be created to successfully test the server.

[7] `http://wiki.basho.com/An-Introduction-to-Riak.html`
[8] `http://couchdb.apache.org`
[9] during the writing of the thesis the percentage went up to 19% `https://github.com/languages/`

3 Implementation of a prototype

This chapter contains the realization of much of the principles and techniques described in chapter 2 on page 6. Here the problems and their solutions are discussed in greater detail, and at times the text becomes more specific to GGS.

Much of what is discussed in this chapter has been implemented in the Erlang GGS prototype. Specific solutions such as *supervisor structures* and distribution of erlang nodes on physical nodes. The different means of communications within the GGS and outside the GGS with third parties are also discussed here.

The chapter ends with case studies and code examples of particular features of the GGS. The case studies and the code serve as concrete examples of the implementations outlined in the rest of this chapter.

3.1 Overview of the prototype

The prototype of the GGS was developed using the Erlang language. The functional and concurrent style of Erlang facilitates development of software based on a real-world model [Armstrong, 2011]. In Erlang, most things are processes. The software running the Erlang code is known as the Erlang machine, or a Erlang node. Each Erlang node is capable of running several *threads* (also known as *Light Weight Processes; LWP*), much like the threads in an operating system. Threads in a Linux system, for example, are treated much like operating system processes in different systems. Due to the size of data structures related to each process, swapping one process for another (known as *context switching*) is an expensive task in many systems [McKusick and Neville-Neil, 2004, pg 80].

The cost of swapping operating system processes becomes a problem when many processes are involved. If the GGS system had been developed using regular operating system processes, it would have had to be designed in a way to minimize the number of processes. Using Erlang, which is capable of running very many processes, several times more than an operating system can, the mapping between the real world system (described in 2.1 on page 6) becomes clearer.

Erlang allows the GGS to create several process for each player connecting, these processes can handle a multitude of different tasks, parsing data for example. Since each task is handled by a different process, the tasks are clearly separated and the failure of one is easily recovered without affecting the others.

In addition to creating (or *spawning*) processes specifically to handle new players connecting, the GGS has more permanent processes running at all times. The constantly running processes in the GGS system are called *modules*. An example of a module in the GGS is the *dispatcher module*, which handles the initial connection made by a client, passing the connection along further in to the system.

In figure 3.1 on the next page the entire GGS system is represented graphically. The circles marked with 'C' topmost in the picture represent game clients. These circles represent processes running on gamers computers, and not on the GGS machine. If a game of chess is to be played on

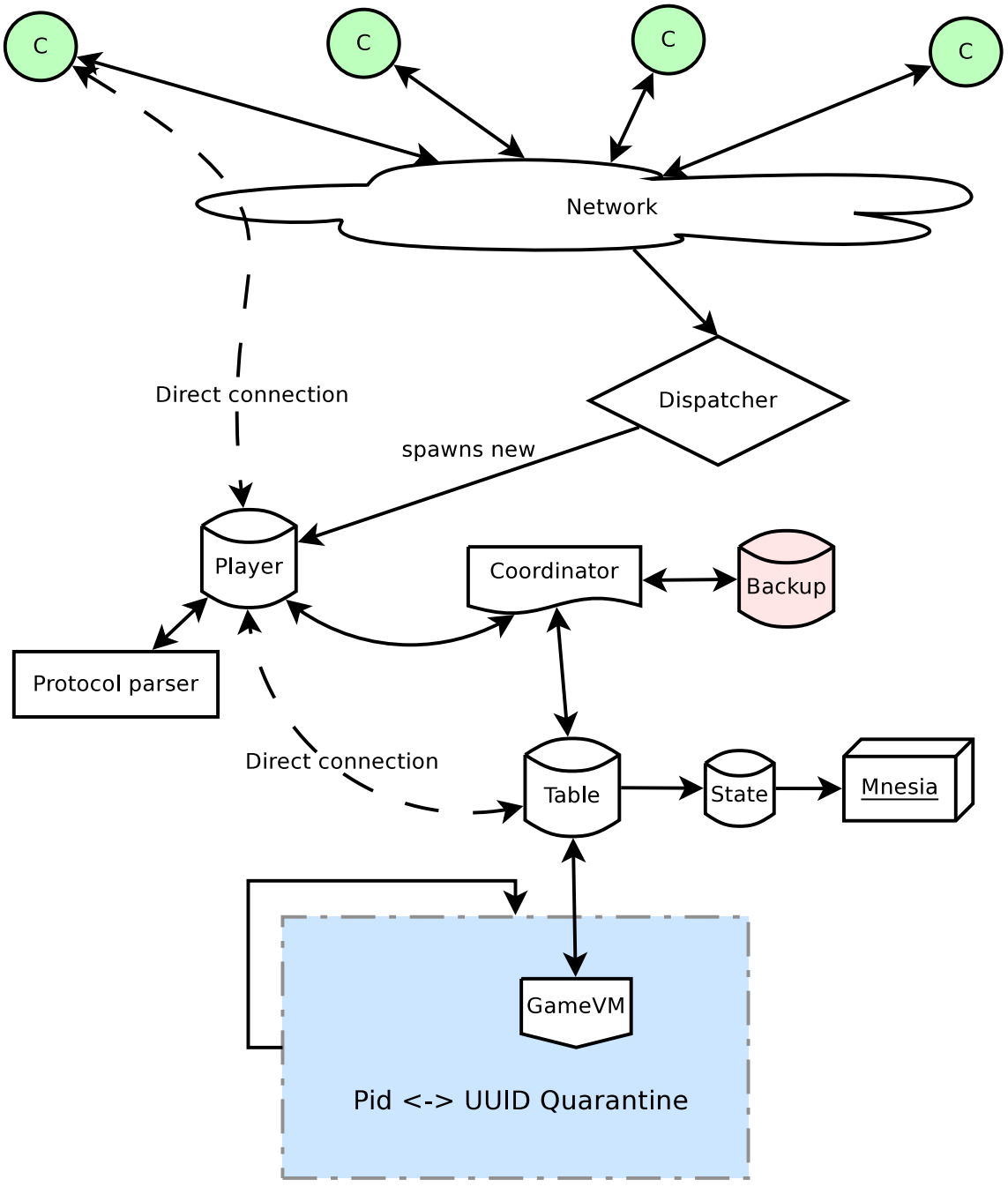

Figure 3.1: The layout of the GGS. The circles marked with 'C' topmost in the picture represent clients. The cloud marked 'network' pictured directly below the clients can be any network, for example the Internet. The barell figure marked 'backup' is a process being fed backup data from the coordinator. The barell marked 'State' contains the state of a table, and this is fed into the box marked 'Mnesia' which is database. Finally the figure shaped as a shield marked 'GameVM' contains the actual game process.

the server, the clients on the gamers machines will be chess game clients. Clients connect through a network, pictured as a cloud, to the dispatcher process in the GGS. The dispatcher process and all other modules are discussed in 3.3 on page 20. For each connection, a new player process is spawned, which immediately after spawning is integrated in to the GGS by the coordinator process.

3.2 The usage of Erlang in the GGS

Erlang was designed by Ericsson, beginning in 1986, for the purpose of creating concurrent applications and improving telecom software. Features essential for the telecom industry to achieve high availability in telecom switches were added to the language.

Erlang uses message passing in favor of shared memory, mutexes and locks, something which at the time was controversial among fellow developers Armstrong [2010]. The reason for using message passing, according to Armstrong, was that applications should operate correctly before optimizations are done, where efficient internal communication within the Erlang machine was considered a later optimization.

In using message passing in favor of the methods commonly used at the time, the issues commonly associated with shared memory and locking were avoided. In Erlang, everything is a process, and everything operates in its own memory space. Memory cannot be shared among processes, which prohibits a process from corrupting the memory of a different process.

Messages are sent between the processes in an asynchronous manner, and each process has a mailbox in which these messages can be retrieved.

Processes in Erlang are also called *Light Weight Processes.* The Erlang processes are very cheaply created. Processes exist within an Erlang machine, or Erlang node. The Erlang machine has its own scheduler and does not rely on the operating system's scheduler, this is a main reason of Erlang's capability of running many concurrent processes Armstrong [2003].

The strong isolation of Erlang processes make them ideal for multi-core and distributed systems. Distribution of software is included as a fundamental part in the Erlang language. The 'physical' location of a process, e.g. which computer the process runs on, is not important when communicating with the process. Processes can communicate regardless of whether they run on the same system of not, transparently.

The distributed nature of Erlang is something the GGS makes use of when scaling across several computers in order to achieve higher performance. The distribution is also important in creating redundancy. Erlang promotes a non-defensive programming style in which processes are allowed to crash and be restarted in favor of having the processes recover from errors. The distributed nature of Erlang means supervisor processes (discussed in section 3.5.1) can reside on remote systems, thereby increasing the reliability of the system as a whole.

A very important feature of Erlang, used in the GGS, is the ability to interface with external hardware and software. Erlang allows communication with external resources through *ports* and *NIF*s (Native implemented functions). Through ports communication can take place much in the same way communication is performed over sockets. NIFs are called like any other functions without any difference to the caller but are implemented in C.

The GGS uses Erlang ports for generating UUIDs[1] and NIFs for interfacing with the virtual

[1] UUIDs are discussed in section 2.7.2

machines of games[2].

Development of the GGS would have been hard if not impossible had it not been for the *OTP* supplied with the standard Erlang distribution. The OTP (Open Telecom Platform) is a set of standard libraries and design patterns, called *behaviors*, which are used when developing Erlang systems.

The GGS makes heavy use of the behaviors supplied in the OTP. The behaviors impose a programming style suitable for distributed and concurrent applications, perfectly suitable for the GGS. In particular, the GGS uses the following behaviors:

- The *supervisor* behavior, which is used when creating a supervisor. Supervisors are used when monitoring processes in the Erlang system. When a process exits wrongfully, the supervisor monitoring the process in question decides which action to take. In the GGS, the most common action is simply to restart the faulting process. A more thorough discussion on supervisors can be found in section 3.5.1.

- The *gen_tcp* behavior, which is used to work with TCP sockets for network communication. Using the gen_tcp behavior, network messages are converted to internal Erlang messages and passed to a protocol parser, where the messages are processed further.

- The *gen_server* behavior, which is used when constructing OTP servers in Erlang. Using this behavior, a state can easily be kept in a server process, greatly increasing the usefulness of the server process. There are many gen_servers in the GGS, it is the most widely used behavior in the project. In addition to introducing a state to the server, the gen_server behavior also imposes patterns for synchronous and asynchronous communication between other gen_servers and other OTP behaviors.

- The *gen_fsm* behavior is used in the protocol parser module in the GGS. Using the gen_fsm behavior, finite state machines are easily developed. Protocol parsers are an ideal example of where to use finite state machines, which are widely used for parsing strings of text.

In addition to supplying behaviors, the OTP also has a style for packaging and running Erlang applications. By packaging the GGS as an *application* the GGS can be started in a way uniform to most erlang software, providing familiarity for other Erlang users, and eases the incorporation of the GGS in other applications.

3.2.1 Short introduction to the Erlang syntax

In order to understand examples in this thesis, a small subset of Erlang must be understood. In this section, the very syntactic basics of Erlang are given.

- **Variables** start with an uppercase letter, examples include `X, Var`, and `Global`. A variable can only be assigned once.

- **Atoms** start with lower case letters, for example: `atom, a`.

[2]Virtual machines of games are discussed in section 2.9

- **Functions** are defined starting with an atom for the name, parenthesis containing parameters, an arrow, a function body and finally a dot marking the end of a function. `square(X) -> X*X.` is an example of a function producing the square of X.
- Functions are **called** by suffixing an atom with the function name with parenthesis, for example `square(10)`. Qualified names can be specified using ':', for example: `math:square(10)`.
- **Tuples** are containers of fixed type for Erlang data types. They are constructed using curly brackets, for example: `{atom1, atom2, atom3}`.
- **Lists** are constructed using [and], for example: `[1,2,3]`.
- **Strings** doubly quoted lists of characters, for example `"Hello world"`.
- **Records** are erlang tuples coupled with a tag for each tuple element. This allows referring to elements by name instead of by position. An example of a record looks like this: `#myRecord`.

3.3 The modular structure of the GGS prototype

The separation of concerns, and principle of single responsibility [3] are widely respected as good practices in the world of software engineering and development. By dividing the GGS up into modules each part of the GGS can be modified without damaging the rest of the system.

The responsibility and concern of each module comes from the responsibility and concern of the real-world entity the model represents. The modeling of the GGS after a real world system was discussed in chapter 2 on page 6.

In the text below the word module refers to the actual code of the discussed feature, while the word process is used when referring to a running instance of the code.

3.3.1 The dispatcher module

The dispatcher module is the first module to have contact with a player. When a player connects to the GGS, it is first greeted by the dispatcher module, which sets up an accepting socket for each player. The dispatcher is the module which handles the interfacing to the operating system when working with sockets. Operating system limits concerning the number of open files, or number of open sockets are handled here. The operating system limits can impose problems on the GGS, this is discussed more in detail in chapter 4 on page 34.

Should the dispatcher module fail to function, no new connections to the GGS can be made. In the event of a crash in the dispatcher module, a supervisor process immediately restarts the dispatcher. There exists a window of time between the crashing of the dispatcher and the restarting of the dispatcher, this window is very short, and only during this window is the GGS unable to process new connection requests. Due to the modular structure of the GGS, the rest of the system is not harmed by the dispatcher process not functioning. The process does not contain a state, therefore a simple restart of the process is sufficient in restoring the GGS to a pristine state after a dispatcher crash.

Returning to scenario of the chess club, the dispatcher module is the doorman of the club. When a player enters the chess club, the player is greeted by the doorman, letting the player in to

[3] More information on the SRP is available at: http://www.objectmentor.com/resources/articles/srp.pdf

the club. The actual letting in to the club is in the GGS represented by the creation of a player process discussed in 3.3.2. The newly created player process is handed, and granted rights to, the socket of the newly connected player.

3.3.2 The player module

The player module is responsible for representing a player in the system. Each connected player has its own player process. The player process has access to the connection of the player it represents, and can communicate with this player. In order to communicate with a player, the data to and from the player object must pass through a protocol parser module, discussed in 3.3.3. Raw communication, without passing the data through a protocol parser is in theory possible, but is not useful.

In the creation of a player process, the coordinator process, discussed in 3.3.4 on the next page, is notified by the newly connected process.

In the event of a crash in a player process, several things happen.

1. The player process, which is the only process with a reference to the socket leading to the remote client software, passes this reference of the socket to the coordinator process temporarily.

2. The player process exits.

3. The coordinator spawns a new player process, with the same socket reference as the old player process had.

4. The player process resumes operation, immediately starting a new protocol parser process, and begins to receive and send network messages again.

The window of time between the crash of the player process and the starting of a new player process is, as with the dispatcher, very short. Since the connection changes owners for a short period of time, but is never dropped, the implications of a crash are only noticed, at worst, as choppy gameplay for the client. Note however that this crash recovery scheme is not implemented in the GGS prototype.

Moving back to the real world example, the player process represent an actual person in the chess club. When a person sits down at a table in the chess club, the person does so by requesting a seat from some coordinating person, and is then seated by the same coordinator. Once seated, the player may make moves on the table he or she is seated by, this corresponds clearly to how the GGS is structured, as can be seen in the following sections.

3.3.3 The protocol parser module

The protocol parser is an easily interchangeable module in the GGS, handling the client-to-server, and server-to-client protocol parsing. In the GGS prototype, there is only one protocol supported, namely the *GGS Protocol*. The role of the protocol parser is to translate the meaning of packets sent using the protocol in use to internal messages of the GGS system. The GGS protocol, discussed below is used as a sample protocol in order to explain how protocol parsers can be built for the GGS.

Algorithm 3.1 A sample packet sent from a client to the GGS during a chat session

```
1 Game-Command: chat
2 Token: e30174d4-185e-493b-a21a-832e2d9d7a1a
3 Content-Type: text
4 Content-Length: 18
5
6 Hello world, guys!
```

The structure of the GGS Protocol

The GGS protocol is modeled after the HTTP protocol. The main reason for this is the familiarity many developers already have with HTTP due to its presence in internet software. Each GGS protocol packet contains a headers section. The headers section is followed by a data section. In the headers section, parameters concerning the packet is placed. In the data section, the actual data payload of the packet is placed.

There is no requirement of any specific order of the parameters in the headers section, however the data section must always follow directly after the headers section.

In the example below, line 1 contains a Game-Command parameter. This parameter is used to determine which game-specific command the client is trying to perform. The handling of this parameter is specific to each game, and can be anything.

Line 2 specifies a game token. This is a UUID which is generated for each client upon authentication with the GGS. The GGS uses this token in case a client is disconnected and the new connection created when the client reconnects must be re-paired with the player object inside the GGS. The UUID is also used as a unique ID within GDL VMs.

Line 3 specifies the content type of the payload of this particular packet. This parameter allows the GGS to invoke special parsers, should the data be encoded or encrypted. When encryption is employed, only the payload is encrypted, not the header section. This is a scheme which does not allow for strong encryption, but is deemed feasible for gaming purposes.

Line 4 specifies the content length of the payload following immediately after the headers section.

The parser of the GGS protocol implemented in the GGS prototype is designed as a finite state machine using the gen_fsm behavior. When a full message has been parsed by the parser, the message is converted into the internal structure of the GGS messages, and sent in to the system from the protocol parser using message passing.

3.3.4 The coordinator module

The coordinator module is responsible for keeping track of all players, their seats and tables. Players register with the coordinator process when first connecting to the server, and the coordinator places each player by their respective table.

The coordinator keeps mappings between each player and table, therefore it is used to perform lookups on tables and players to find out which are connected. The connectivity of players and tables is important when sending messages to all participants in a game. A lookup in the coordinator process is performed prior to notifying all players in a game to ensure the message reaches all players. The lookup can be performed either using internal identification codes or using the

UUID associated with each client and table.

The coordinator process contains important state, therefore a backup process is kept at all times. All good data processed by the coordinator is stored for safekeeping in the backup process as well. Data which is potentially harmful is not stored in the backup process.

Upon a crash, the coordinator process recovers the prior good state from the backup process and continues where it left off. A supervisor process monitors the coordinator process and restarts the process when it malfunctions. There is a window of time between the crash of the coordinator and the restarting of the coordinator, during this time, players cannot be seated by new tables, and cannot disconnect from the server. This window of time is very small, and the unavailability of the coordinator process should not be noticed by more than a short time lag for the clients.

Moving back to the example of the chess club, the coordinator process can be seen as a judge, monitoring all moves of the players. At the same time as acting as a judge, the coordinator process is also a host in the chess club, seating players by their tables and offering services to the players.

3.3.5 The table module

The table module is mostly a hub used for communication. New table processes are created by the coordinator on demand. The table module does not contain any business logic, however each process contains information concerning which players are seated by that particular table.

The information about which players are seated by each table is used when notifying all players by a table of an action. Consider a game of chess, each player notifies the table of its actions, the table then notifies the rest of the participants of these actions after having had the actions processed by the game VM, where an action could be moving a playing piece.

Each table is associated with a game VM. The actions sent to a table are processed by the game VM, this is where the game logic is implemented.

After a crash in a table process, the entire table must be rebuilt and the players must be re-associated with the table. Data concerning players is kept in the coordinator process, and is restored from there. Data kept in the actual game is not automatically corrupted by the crash in a table, however the table must be re-associated with the game VM is was associated with prior to the crash of the table. The table process maps well into the setting of the real-world chess club scenario previously discussed. A table works in the same way in a real world setting as in the GGS setting.

3.3.6 The game virtual machine module

This module holds the game logic of a game and is responsible for the VM associated with each game.

The game VM contains the state of the VM and a table token associated with a running game. GameVM is started by the table module. The table module hands over a token to the game VM during initialization. During initialization a new VM instance and various objects associated to the VM instance will be created. Callbacks to Erlang are registered into the VM and then the source code of a game is loaded into the VM and the game is ready for startup. The only means for a game to communicate with the VM is through usage of a provided interface.

The VM itself makes it possible for the game developer to program in the programming language covered by the VM. In future releases, more game VMs will be added to support more programming

languages. Because the game VM keeps track of the correct table, the game developer does not need to take this into consideration when programming a game. If a method within the game sends data to a player, it will be delivered to the player in the correct running game. The same game token is used to store the game state in the database. Therefore, no game states will be mixed up either.

This module does not affect game runtime but evaluates a new game state and handles communication between the game and the players. A closer look at the structure of this model is given in 3.4 on the following page.

The code which is run in the VM is uploaded to the GGS prior to each game. Allowing the clients to upload code allows clients to run any game.

3.3.7 The database module

Game data from all games on the GGS are stored in the database backend of the database module.

In the GGS prototype the database module is using a database management system called Mnesia. Mnesia ships with the standard Erlang distribution and is a key-value store type database. Mnesia is designed to handle the stress of telecoms systems, and has some features specifically tailored for telecoms which are not commonly found in other databases. Key features of the Mnesia database are:

- Fast key/value lookups
- Distribution of the database system
- Fault tolerance

Mattsson et al. [1998]

The features of Mnesia originally intended for telecoms prove very useful for the purposes of the GGS as well. The fault tolerance and speed of Mnesia are very valuable tools, the fast key/value lookups permit many lookups per second to the database.

Game data will not be lost when a game is stopped or has gone down for unknown reasons. This makes it possible to continue a game just before the failure without having to start the game from the beginning.

The GGS stores the game state in the distributed Mnesia database, from which the state can be restored in the event of a crash.

Each game is uniquely identified by a table token and the data of each game is stored within two different namespaces. The namespaces are named World and Localstorage. The World is used contain all game data related to the game state. This sort of game data may change during the runtime of the game. The Localstorage should contain data independent of the game state. Game resources, constants and global variables are all examples of data that could reside within the Localstorage. To store a value within the database, not only is the table token and the name of the namespace required, but a unique key so that the value can be successfully retrieved or modified later. The key is fully decidable by the game developer.

The interface of the database module is an implementation of the upcoming W3C Web Storage specification. Web Storage is intended for use in web browsers, providing a persistent storage

on the local machine for web applications. The storage can be used to communicate in between browser windows (which is difficult when using cookies), and to store larger chunks of data Hickson [2011]. Usage of the web storage standard in the GGS provides a well documented interface to the database backend.

3.4 Communication with the GDL VM

A game launched on the GGS is run within a virtual machine. For each programming language supported, there is a virtual machine that interprets the game. Furthermore an interface for communication between the GGS, the game and the players playing the game is present.

Callbacks written in Erlang are registered to the VM for the interface to work. It is only with the help of the interface that the game developer can access the game state and send messages to the clients. The interface provides access to three objects called *world, players* and *localStorage*. The game state is safely stored in a database and retrieved for manipulation by a call for the world object. Interaction with the players is done by using the *GGS.sendCommand(player_id, command, args)* and *GGS*.sendCommandToAll(command, args). The localstorage is a convenient way to store global data and other variables separated from the game state. Unique ids called gametokens are generated for hosted games so that they are not mixed up.

A game launched on the GGS is run within a virtual machine. For each programming language supported, there is a virtual machine that interprets the game. Furthermore an interface for communication between the GGS, the game and the players playing the game must be present.

3.4.1 Exposing Erlang functionality to the GDL VM

This section contains a concrete example of how the localstorage and world objects are exposed to a GDL VM. The example comes from the GGS prototype, which uses JavaScript powered by Google V8 as its GDL VM.

The code given in 3.2 is specific to V8 and JavaScript, however implementations for different GDLs, or different VMs should be similar.

In JavaScript is is common to use a top level object, called a global object, to establish a global scope. This allows the declaration of global variables and functions. To gain access to the global object in the GGS, the `erlv8_vm:global(..)` function on line 2 of the example is used. Using the global object, declarations of the world and GGS object can be placed in the global scope.

`Global:set_value(..)` is a call to the global object, declaring new objects in the global scope. On line 4 the GGS object is declared. By accessing `GGS.localStorage` from within the GDL, access to the localstorage is provided, thus the localstorage must be connected to the GGS object, this can be seen in line 5.

Both the GGS and localstorage objects are dummy objects, which provide no functionality, these two objects are simply placed in the GDL for the purpose clearing up the code. In order to perform an action using the GGS and localstorage objects, the `getItem` and `setItem` functions must be used. These items are directly connected to the database module of the GGS, which is discussed in more detail in 3.3.7.

Similarly the functions `sendCommand, sendCommandToAll` and `setTimeout` are directly connected to a piece of code in the GGS which performs the desired action. The `sendCommand` func-

Algorithm 3.2 An example of how Erlang functionality is exposed to a JavaScript GDL

```
% @doc Exposes some GGS functions to JavaScript
expose(GameVM, Table) ->
    Global = erlv8_vm:global(GameVM),
    Global:set_value("GGS", erlv8_object:new([
        {"localStorage", erlv8_object:new([
            {"setItem", fun(#erlv8_fun_invocation{}, [Key, Val])->
                ggs_db:setItem(Table, local_storage, Key, Val)
            end},
            {"getItem", fun(#erlv8_fun_invocation{}, [Key])->
                ggs_db:getItem(Table, local_storage, Key)
            end}
            % more functions ...
        ])},
        {"world", erlv8_object:new([
            {"setItem", fun(#erlv8_fun_invocation{}, [Key, Val])->
                ggs_db:setItem(Table, world, Key, Val),
                ggs_table:send_command_to_all(
                  Table, {"world_set", Key ++ "=" ++ Val}
                )
            end},
            {"getItem", fun(#erlv8_fun_invocation{}, [Key])->
                ggs_db:getItem(Table, world, Key),
            end}
            % more functions ...
        ])},
        {"sendCommand", fun(#erlv8_fun_invocation{}, [Player, Command, Args])->
            ggs_table:send_command(Table, Player, {Command, Args})
        end},
        {"sendCommandToAll", fun(#erlv8_fun_invocation{}, [Command, Args])->
            ggs_table:send_command_to_all(Table, {Command, Args})
        end}
         {"setTimeout", fun(#erlv8_fun_invocation{}, [Time, Function])->
            timer:apply_after(Time, ?MODULE, call_js, [GameVM, Function])
        end}
        % more functions ...
    ])).
```

tions are used to send commands or text to participants of the table. The `setTimeout` function introduces timeouts to the V8 engine, which are not available per default.

3.5 Techniques for ensuring reliability

One of the main goals of the project is to achieve high reliability. The term “reliable system” is defined by the IEEE as a system with “the ability of a system or component to perform its required functions under stated conditions for a specified period of time” Electrical and [ieee]. There are some tools for creating reliable applications built in to Erlang.

- Links between processes. When a process spawns a new child process, and the child process later exits, the parent process is notified of the exit.
- Transparent distribution over a network of processors. When several nodes participate in a network, it does not matter on which of these machines a process is run. Communication between processes does not depend on the node in which each process is run.
- Hot code replacements. Two versions of the same module can reside in the memory of Erlang

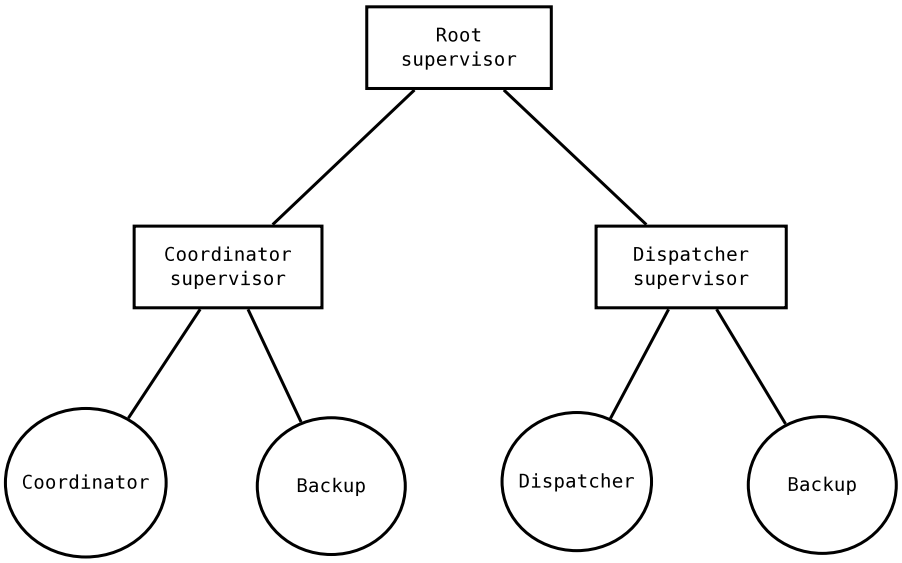

Figure 3.2: The supervisor structure of GGS

at any time. This means that a simple swap between these versions can take place very quickly, and without stopping the machine.

These three features are some of the basic building blocks for more sophisticated reliability systems in Erlang. Many times it is not necessary to use these features directly, but rather through the design patterns described below.

3.5.1 Supervisor structure

By linking processes together and notifying parents when children exit, supervisors are created. A supervisor is a common approach in ensuring that an application functions in the way it was intended Savor and Seviora [1997]. When a process misbehaves, the supervisor takes some action to restore the process to a functional state.

There are several approaches to supervisor design in general (when not just considering how they work in Erlang). One common approach is to have the supervisor look in to the state of the process(es) it supervises, and let the supervisor make decisions based on this state. The supervisor has a specification of how the process it supervises should function, and this is how it makes decisions.

In Erlang, we have a simple version of supervisors. We do not inspect the state of the processes being supervised. We do have a specification of how the supervised processes should behave, but on a higher level. The specification describes things such as how many times in a given time interval a child process may crash, which processes need restarting when crashes occur, and so forth.

When the linking of processes in order to monitor exit behavior is coupled with the transparent distribution of Erlang, a very powerful supervision system is created. For instance, we can restart a failing process on a different, new node, with minimal impact on the system as a whole.

In the GGS, we have separated the system in to two large supervised parts. We try to restart

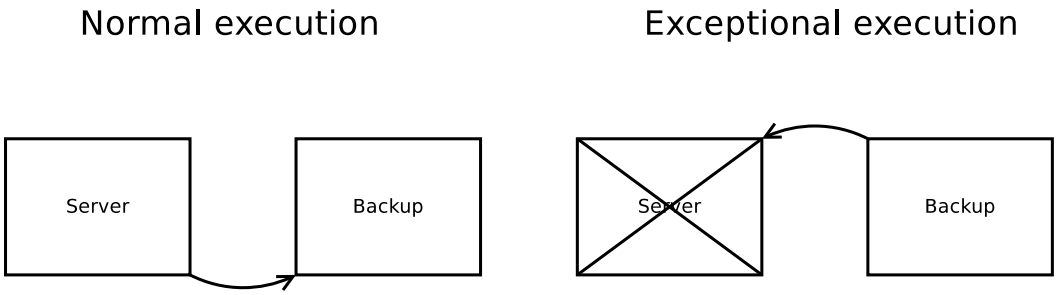

Figure 3.3: To the left normal execution is pictured; the server state is backed up. To the right; the exceptional execution, where the state is retrieved from the backup to repopulate the server.

a crashing child separately, if this fails too many[4] times, we restart the nearest supervisor of this child. This ensures separation of the subsystems so that a crash is as isolated as possible.

The graphic above shows our two subsystems, the coordinator subsystem and the dispatcher subsystem. Since these two systems perform very different tasks they have been separated. Each subsystem has one worker process, the coordinator or the dispatcher. The worker process keeps a state which should not be lost upon a crash.

We have chosen to let faulty processes crash very easily when they receive bad data, or something unexpected happens. The alternative to crashing would have been to try and fix this faulty data, or to foresee the unexpected events. We chose not to do this because it is so simple to monitor and restart processes, and so difficult to try and mend broken states. This approach is something widely deployed in the Erlang world, and developers are often encouraged to "Let it crash".

To prevent any data loss, the good state of the worker processes is stored in their respective backup processes. When a worker process (re)starts, it asks the backup process for any previous state, if there is any that state is loaded in to the worker and it proceeds where it left off. If on the other hand no state is available, a special message is delivered instead, making the worker create a new state, this is what happens when the workers are first created.

3.5.2 Redundancy

The modules in the GGS are built to be capable of redundant operation. By adding a backup process to sensitive processes, the state can be kept in the event of a crash. The coordinator of the GGS prototype has this backup feature built in. The coordinator passes state along to the backup process which keeps the data safe. In the event of a crash, the coordinator recovers the state from the backup process. Figure 3.3 depicts the redundancy built in to the coordinator process.

3.5.3 Hot code replacement

Hot code replacement is a technique used to update systems while they are running. The main use of hot code replacement are in critical systems that require low downtime, such as telecom systems. By using hot code replacement system can be able to achieve as high uptime as possible and thus improving the reliability of the system. Code replacement is a feature that exist in Erlang which means that with some work it could be implemented into the GGS.

[4]Exactly how many "too many" is depends on a setting in the supervisor, ten crashes per second is a reasonable upper limit.

3.6 Software testing

In order to make sure the GGS prototype adheres to the specification set two different approaches to software testing are used. For simpler testing the GGS prototype uses unit tests. Modules are tested on a high level, making sure each function in the module tested functiions according to specification.

Unit testing is not employed to test the system from the client side. In order to more accurately simulate real users some randomization is needed, as users do not always act rationally. In order to introduce random data, the client side of the GGS is simulated by QuickCheck tests.

3.6.1 Unit testing

Unit testing is a way to check if the functionality adheres to the specification of the system by manually creating test cases for sections of code. In most cases whole functions. Unit testing is good, not only for revealing software bugs, but also to state that a feature is working according to the specification. Unit testing is a common way to test software and has proven useful within the GGS when functions take complicated arguments. In these cases it is easy to set up a scenario that should work.

Unit testing is a useful way to create regression tests. Regression tests are used to make sure changes made to the GGS do not introduce new bugs or break the specification. The regression tests are optimally run very often, such as after each change to the code.

3.6.2 Automated test case generation

The problem of writing software tests manually, is that it takes a lot of time. There exists other ways to test software that address this problem by generating test cases with certain properties. This allows for testing functions with a lot of different input parameters without having to implement each specific test itself.

By having each test automatically generated, each test can be very complex and long. In order to generate random, complex tests the GGS uses QuickCheck. By using QuickCheck the GGS can be tested with input which would be extremely difficult to construct using manual testing methods. Regression tests, such as the unit tests used by the GGS are more useful for ensuring the system does not diverge from a working scenario than for finding new cases where the specification does not hold Arts et al. [2006].

The entire GGS was not tested using QuickCheck, nor was the entire client protocol for a game tested using QuickCheck, however the tests performed using QuickCheck show that an automated testing system such as QuickCheck is a very viable testing method for the GGS.

QuickCheck has features to generate very large and complex tests, the results of which can be hard to analyze. The solution to reading these complex test is to extract a *minimal failing test case* which contains the smalles failing test sequence. By applying a very large test and gradually simplifying the test to find the smallest failing sequence, many bugs which would other wise have been hard to catch can be caught Arts et al. [2006].

QuickCheck was originally made for the programming language Haskell. There are a lot of reimplementations of QuickCheck in various programming languages. Erlang QuickCheck (EQC) and Triq are two variants of QuickCheck for Erlang. EQC was chosen for testing the GGS. Besides

the standard functionality that QuickCheck provides, EQC is capable of testing concurrency within a program.

3.7 Case studies

This section contains three case studies. These case studies have been written to provide examples of how the flow through the GGS can look when performing different tasks. The first case study outlines the flow of sending a common message to the GDL VM and receiving a response. The second case study provides an example of the process of connecting to the GGS to set up a game. The third and final case study is a section of code from a part of a game for the GGS. The code in the third study shows how a simple chat server can be implemented in the GGS using JavaScript as GDL.

3.7.1 Typical communication

This case study describes the flow through the GGS when a typical command is encountered. Below is a case study where a chat client sends a message to change the nick of a user. The actual code performing the change of a nick in JavaScript is discussed in section 3.7.3. All communication between modules is asynchronous, nothing is blocking, which is very important in concurrent systems. To follow the example more easily, looking at the graphic in section 3.1 on page 17 is recommended.

1. The client packages a Game-Command into a *GGS protocol packet* which conforms to the protocol structure the GGS is using and sends it over the network.

2. The player process, which is coupled to the TCP-process which reacts on incoming messages, accepts the message and forwards the raw data to the protocol parser process.

3. The protocol parser process parses the message and brings it into the format of the internal GGS presentation of such a message, which is just a specialized Erlang tuple.

4. The protocol parser sends this Erlang touple back to the player process.

5. The player process checks if it is a Server-Command or a Game-Command. In our example it is a Game-Command and it sends the message to the table process.

6. The table process sends it to its own Game VM process.

7. The game VM process calls the function *playerCommand("278d5002-77d6-11e0-b772-af884def5349", "nick", "Peter")* within the JavaScript VM.

8. The JavaScript VM (JSVM) - at this stage Googles V8 JavaScript Engine - evaluates the function within the sandboxed game context which has been established earlier during the setup of the game.

9. In the example in section 3.7.3 we see that the GGS-functions *GGS.localStorage.setItem(key, value)* and *GGS.localStorage(key)* are used. Both are callbacks coupled to the database module functions.

10. Data is being read from and written to the database and handed over to the JSVM via the database process.

11. In the example the *GGS.sendCommandToAll()* is being called then which is a callback to a function of the table module which iterates through its player list and sends the command to every player.

12. The table process sends every player process the message to send the message with the change of a nickname of a particular user to its own client.

13. The player process asks the protocol process to create a message conforming to the protocol which is being used.

14. The protocol process creates a string according to the protocol and returns it to the player process.

15. The player process sends the message with help of the gen_tcp module to the client.

3.7.2 Initialization and life cycle of a game

This case study describes the initialization and definition of a game and in roughly its life cycle untill it is removed from the GGS.

Initialization

1. A client connects via TCP to the GGS.

2. The dispatcher process reacts on the incomming connecction and creates a new player process.

3. The dispatcher process couples the TCP connection to the newly created player process, this way the new player process is responsible to react on incoming messages.

4. The client sends a message with a HELLO Server-Command to initiate a handshake.

5. The player module parses the message with help of the protocol module.

6. If the message was just a plain HELLO, without a table token, then the player process asks the coordinator process to create a new table process and add this player process to this newly created table. If the client did send a table token then the player process asks the coordinator to att the player process to this table.

7. During the creation of a new table the table process creates a new game VM process which creates its own game context within the JavaScript VM.

8. The player process answers to the client with a HELLO Client-Command and passes on the clients player token along with the information about if it should define a game - because it is the first client to connect to this table - or not and the table token it was assigned to.

Defining a game

The generic nature of the GGS leaves it up to the client to define which game should be run. The definition is done in the GDL, in this example, the GDL is JavaScript. It is possible to alter the GGS prototype so that only the server maintainer is able to install new games on the server however the current implementation of the GGS is much more generic.

The first client which connects to a table is responsible to provide the JavaScript server source code. To do so there is a DEFINE Server-Command.

1. If during the handshake with the HELLO command the client is assigned the task of providing the server source code then the client must send a DEFINE Server-Command message with the source code as its parameter. Only the first client will get the information about the need of defining a game during the handshake.
2. The player process parses the message, with help of the protocol module.
3. The player process sends the source code to the table process assigned to the player as a DEFINE message.
4. The table process forwards the source code to the game VM process.
5. The game VM process executes the source code within the JavaScript VM.
6. The JavaScript VM evaluates the source code - which has to implement the playerCommand() function - within the context of the game.
7. The game is at this point fully initialized and can be used by all clients with help of the playerCommand() function.
8. The table process saves the source code in the database for backup reasons (this is not yet implemented).
9. The player process sends a DEFINED Client-Command to the client. This way the client is notified that everything went well and it can start the game.

Life cycle

1. Initialization
2. Defining a game
3. Other clients connect and initialize but do not define anything.
4. Typical communication
5. Clients disconnect
6. When the last client disconnects the table process terminates and with it the game context and database content (not implemented in the prototype).

Algorithm 3.3 A concrete example of a simple chat server written in JavaScript, running on the GGS

```
1  function playerCommand(player_id, command, args) {
2      if(command == "nick") {
3          changeNick(player_id, args);
4      } else if(command == "message") {
5          message(player_id, args);
6      }
7  }
8  function changeNick(player_id, nick) {
9      var old_nick = GGS.localStorage.getItem("nick_" + player_id);
10     GGS.localStorage.setItem("nick_" + player_id, nick);
11     if (!old_nick) {
12         GGS.sendCommandToAll("notice", nick + " joined");
13     } else {
14         GGS.sendCommandToAll("notice", old_nick + " is now called " + nick);
15     }
16 }
17 function message(player_id, message) {
18     var nick = GGS.localStorage.getItem("nick_" + player_id);
19     GGS.sendCommandToAll('message', nick + "> " + message);
20 }
```

3.7.3 A GGS server application in JavaScript

Below is a concrete example of a simple chat server application written using the GGS. The language chosen for this chat server is JavaScript. The GGS processes all incoming data through a protocol parser, which interprets the data and parses it into an internal format for the GGS.

When the GGS receives a *Game-Command* from a client, it is passed along to the game VM through a function called *playerCommand* which is the entry point for each game and has to be implemented by the developer; one can think of it like the *main()* function of a C or Java program. Typically the *playerCommand* function contains conditional constructs which decide the next action to take. In 3.3 an example of the *playerCommand* function can be seen.

In 3.3 the *playerCommand* function accepts two different commands. The first command is a command which allows chat clients connected to the chat server to change nicknames, which are used when chatting. In order to change the nickname, a client must send a Game-Command "nick" with the actual new nickname as a argument. When a message arrives to the GGS which has the form corresponding to the nickname change, the *playerCommand* function is called with the parameters *player_id, command,* and *args* filled in appropriately.

The *playerCommand* function is responsible for calling the helper functions responsibly for carrying out the actions of each message received. *changeNick* is a function which is called when the "nick" message is received. The *changeNick* function uses a feature of the GGS called localstorage (see section 3.4), which is an interface to the database backend contained in the database module (see 3.3.7). The database can be used as any key-value store, however the syntax for insertions and fetch operations is tightly integrated in the GDL of the GGS.

Access to the localStorage is provided through the *GGS object*, which also can be used to communicate with the rest of the system from the GDL. Implementation specifics of the GGS object are provided in 3.4.

4 Problems of implementation

This chapter contains specific problems encountered when implementing the GGS prototype. Some of the problems described have solutions attached, however some problems were not solved, therefore only ideas for solutions have been attached.

The integration of JavaScript as a GDL in the GGS prototype was particularly difficult, and is handled in this section and so is the protocol design.

4.1 JavaScript engine

The GGS prototype uses a virtual machine to sandbox each game. JavaScript was chosen for the prototype due to its commonality in web applications and the flexibility of the language. Any language with the proper bindings to Erlang could have been used in theory.

There are two JavaScript virtual machines, or *engines,* with suitable bindings to erlang available at the time of the writing of this thesis. There is a group of machines developed by Mozilla called *TraceMonkey, JaegerMonkey, SpiderMonkey* and *IonMonkey*, and also there is Googles *V8*. The members in the group of Mozilla machines are largely the same, and are referred to as the same machine for simplicity.

For the Mozilla machines, there exists a Erlang binding called erlang_js, and for the V8 machine a binding called erlv8 exists.

4.1.1 erlang_js

erlang_js provides direct communication with the JavaScript VM. Which is exactly what is desired, however also required is the possibility to communicate from JavaScript to Erlang. The ability to communicate from JavaScript to Erlang is not yet implemented in erlang_js, due to lack of time of the erlang_js developers.

There were two possible solutions to the problem, either one would implement the missing functionality, or a switch from erlang_js to some other JavaScript engine with better bindings could be made.

Attempts at implementing the missing functionality were initially made but never became stable enough for usage in the GGS and the erlang_js software was abandoned.

4.1.2 erlv8

erlv8 is powered by the V8 engine developed by Google. The ability to communicate from JavaScript to Erlang using callbacks (aka NIF) is available in the erlv8 bindings and can be used within the GGS.

Initial releases of the erlv8 bindings had stability issues, these however were resolved by the erlv8 developers during the development GGS. At this point erlv8 is the JavaScript engine powering JavaScript as a GDL in the GGS.

4.2 Protocol design

Initially the GGS protocol was planed to use the UDP protocol for transport. Due to the lack of error checking in the UDP protocol, the UDP protocol is faster than the TCP protocol, this was a main reason in the desire to use UDP. The GGS does however need error checking for some of it parts to be as reliable as possible. Therefore an error checking layer would have to be placed on top of UDP.

The development of an error checking layer was weighed against the implementation of TCP instead of UDP, thus losing some speed. Even though speed was lost, TCP was chosen due to the relative ease of implementation compared to UDP. Due to the modularity of the GGS, a UDP extension is easily possible by replacing the network parts of the GGS.

The Apache Thrift [Agarwal et al., 2007] was also an alternative. Using Thrift would mean the GGS would feature a standard protocol for network communication. Before finding out about Thrift during a lecture of Joe Armstrong (one of the inventors of Erlang), an implementation of the GGS protocol had already been implemented, moving to Thrift would mean too much efford for a prototype during the short amount of time.

The use of Thrift, Google protocol buffers - which is a different approach to that implemented by Google - or other protocols can be supported quite easily by developing protocol modules for each the protocols. No protocol modules for these protocols have however been developed during the writing of this thesis.

5 Results and discussion

In this chapter the results of the GGS prototype are presented and discussed. The results of the ing are presented with both graphical and textual content. Finally thoughts about how future improvements to the prototype could look like are given.

5.1 Statistics

Testing of the GGS took place in two separate sessions. The first session simulates a highly demanding application, the second session simulated a less demanding application. The highly demanding application is a real time game which does several asynchronous database writes each second. The less demanding application does not perform any database reads or writes.

Each of the two simulations use JavaScript as the GDL. The JavaScript is run through Google V8. The database module uses Mnesia.

During the sessions two measurements were recorded.

- **Messages per second** is used to see how many incoming and outgoing messages the server can process each second. The results of the messages per second testing are shown for a high demanding application in figure 5.1, and for a low demanding application in 5.3.
- **Latency between server and client** is used to measure the round-trip time for a message travelling between the client and server. This measurement is used to determine how many players the server can handle while still providing a playable gaming experience. The results of the latency test can be seen in figure 5.2.

The hardware that the GGS was running on was a Thinkpad T410, with a Intel i5 processor and 4GB of RAM.

In the first test, where Mnesia was used, the server had a peak value of nearly 6000 messages per second. When this number was reached Mnesia warned that it was overloaded and shortly after that Mnesia failed to serve requests. This result was not unexpected as this test put the database under heavy load. In the next testing session, the test was conducted with another client that did not use Mnesia. Without mnesia the server peaked at 60000 messages per second, however this was only for a very short time. The average throughput was around 25000 messages per second, five times more than what the server was able to process with Mnesia in place.

In the second testing session the delay between the server and clients was also measured. A connection can be seen between those values, as long as the server is under moderate load the delay is low and stable. When the load on the server increases heavily the delay does the same, this is because the server cannot process all incoming messages and therefore messages are put in a queue within the system.

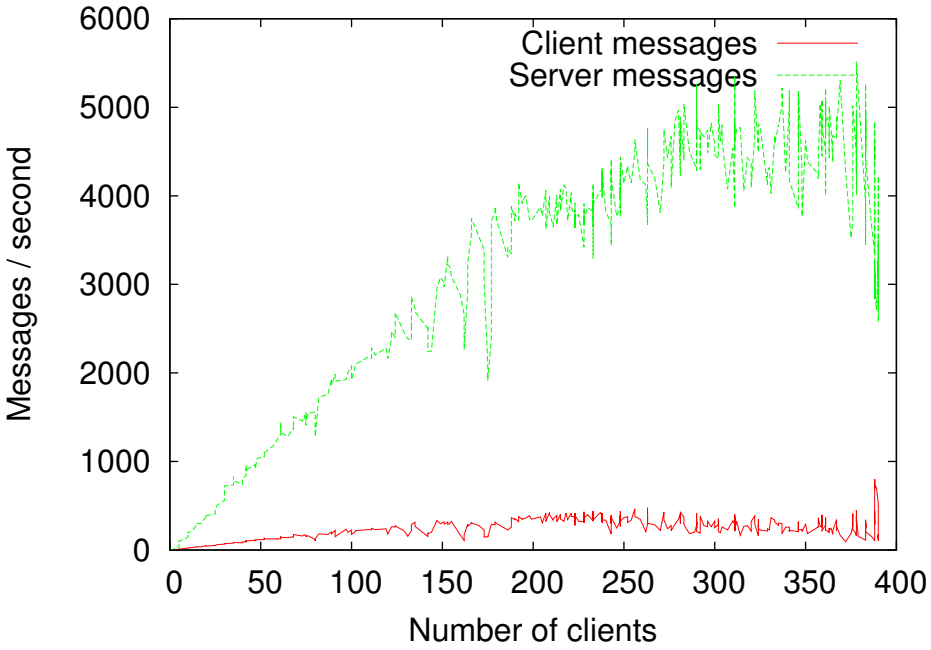

Figure 5.1: The graph shows messages per second for intervals of clients connected. Each client performs 3 asynchronous writes to the Mnesia database each second.

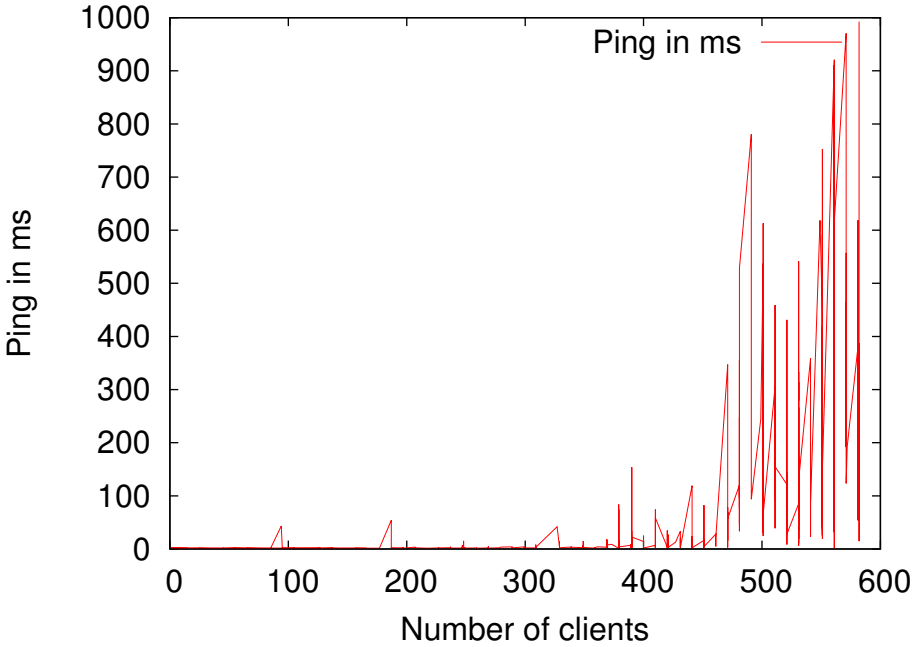

Figure 5.2: This graph shows the latency in a low-demand application. The ping is measured in milliseconds for a message to make a round-trip between client and server.

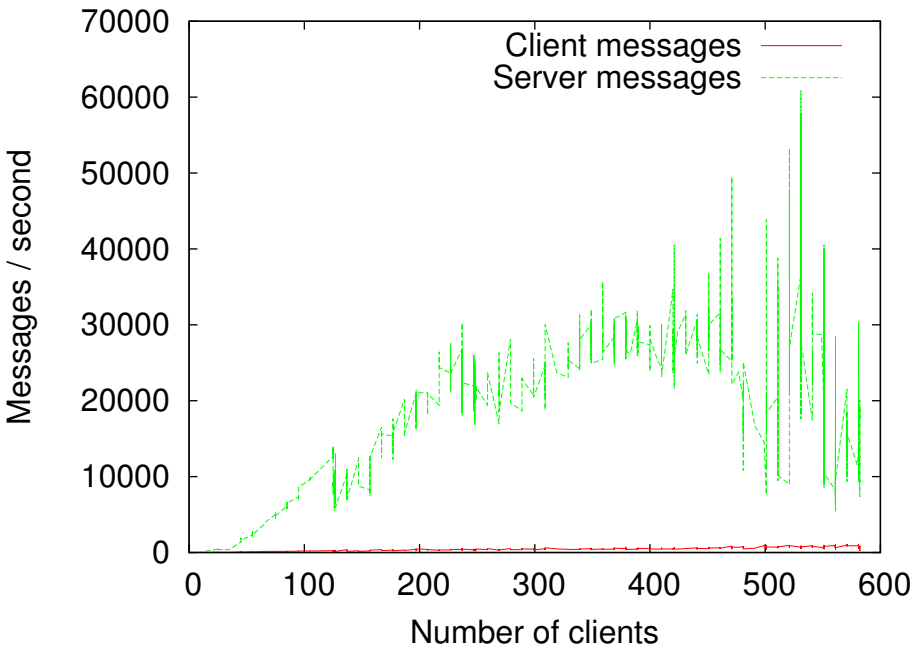

Figure 5.3: The graph shows messages per second for intervals of clients connected. No database is connected.

5.2 Future improvements

There are several things in the GGS that can be improved. In this section the most important additions to the GGS are described, along with a motivation as to why these additions are not found in the GGS prototype.

5.2.1 Distribution

The GGS was originally intended to be a distributed application, running on several machines at once. The design of the GGS should support this, it has however not been tested. The technologies, such as supervisor trees and the servers supplied by the OTP which are used in the GGS all support the development of distributed applications.

Distribution was however not implemented in the GGS. Other parts of the GGS were prioritized. A futute improvement is therefore to implement distribution in the GGS. A simple way to achieve this is to keep one GGS instance as a coordinating instance, and to keep clients on other instances of the GGS, which can be dynamically added as new clients connect.

5.2.2 Performance

The GGS prototype was not developed for maximum performance. Performance optimizations were considered, many were however not implemented in the prorotype. There are several performance optimizations which can be included in future versions of the GGS, below are some of the most important performance optimizations identified.

Protocols

Because of TCP being a connection oriented protocol, it is not suited for all types of game data transfers. Each transmission will consume more network bandwidth than connectionless protocols like UDP and cause unnecessary load on the processor. Therefore support for UDP would mean that more games could be run simultaneously on the GGS. Another advantage of UDP is latency being reduced. Without having to setup a connection for each group packets of data being sent, they will be sent instantly and therefore arrive earlier. Latency is of highest importance in real-time games as it improves realism and fairness in gameplay and many game developers require the freedom to take care of safety issues as packet losses themselves. This concludes that UDP would be a benefit for the GGS, game developers and players alike.

Database

Currently Mnesia is used for game data storage. During stress tests, Mnesia has turned out to be the bottleneck due to data losses when too many games are played on the GGS simultaneously.

The usage of Mnesia in the GGS is not the usage originally intended. Originally a cache was to be placed before Mnesia. The cache could be either Erlang Term Storage (ETS) or a Erlang process which keeps track of all database actions. The cache periodically flushes its contents to Mnesia, thereby reducing the Mnesia transactions overall.

The cache was never implemented in the prototype due to other parts of the GGS being prioritized. The current implementation of the database backend is not optimal, however it functions reliably, therefore it was deemed sufficient for the prototype.

A possible future addition to the GGS could be to add this cache in the database module. The API would not need to change, as this could be implemented internally in the database module.

5.2.3 Documentation

To start the GGS is not self explanatory. This together with overall usage of GGS should be documented. The interface for usage of game developers are also in need of documentation. Features and requirements with respect to the GGS would assist users to know what they need to use the GGS and how they would benefit of it. The GGS does not support many programming languages nor does it have a complete documentation. This needs to be taken care of in future versions.

6 Conclusion

NetHack An early computer game developed by the NetHack team, arguably the oldest computer game still in development

Pacman An early graphical computer game developed by Namco

Zork A textual computer game developed by students at MIT

.NET Software platform

ActionScript Programming language

Amazon EC2 A cloud computation service

Application A way of packaging Erlang software in a uniform way

AXD301 Telephone switch developed by Ericsson

Behaviour A design pattern in OTP

C++ Programming language

COBOL Programming language

Context switch The act of switching from one context, commonly a process, to another. Used by operating systems to achieve multi tasking

CouchDB Database server

Counter-Strike A multiplayer first person shooter game, popular in E-Sports.

Doom A first person shooter series developed by ID software. The series consists of three games.

Downtime The amount of time a system is unavailable and does not function

Erlang A concurrent programming language, often used for telecom applications. The main language of the GGS

ETS Erlang Term Storage

First-person shooter A game in which centers around gun combat from the first person perspective.

Framework A supporting structure, the GGS is a framework for developing network games

GDL Game Development Language, the language used to program games in the GGS

GGS Generic Game Server, a software for reliably hosting network games. The subject of this thesis.

GitHub.com Social coding website

Hardware failiure A failiure in hardware (hard drive, memory, processor, etc) which causes a system to stop functioning

HTTP Hyper Text Transport Protocol, a network protocol commonly used to deliver web pages

IEEE Institute of Electrical and Electronics Engineers, read I-triple-E

JavaScript A programming language originally developed by Netscape, common in web programming

Java Programming language

Latency A measure of delay, often measured in milliseconds

Lua Programming language

LWP Light Weight Process

MAC Address Media Access Control address, used to identify network cards

MMORPG Massively multiplayer online role playing game. An online game with several thousand participants.

Mnesia Database server used in the GGS

Module A part of a larger system

Mutex A construct for achieving mutial exclusion, used to avoid simultaneous access to shared resources in computer systems

Network split Separation of two networks, occurs when two networks cannot communicate, commonly because of a hardware or software failiure

Object Oriented Programming A programming paradigm focusing on objects

OTP Open Telecom Platform, a software suite for Erlang

Quake A first person shooter series developed by ID software. The series consists of four games.

Reliability The ability of a system or component to perform its required functions under stated conditions for a specified period of time

Riak Database server

Sandbox A protected environment in which computer software can be run safely

SHA-1 Cryptigraphic hash function, designed by the National Security Agency (NSA)

Software failiure A failiure in software (the GGS, the operating system, etc) which causes a system to stop functioning

SpiderMonkey JavaScript engine developed by Mozilla

SQL Structured Query Language, a computer language common in querying certain databases

SRP Single Responsibility Principle

Supervisor A process monitoring and hadning crashes in other processes

TCP Transmission Control Protocol, a streaming network protocol

The nine nines A common goal for availability in the telecom business. A system with nine nines of availability is available 99.999999999

UDP User Datagram Protocol, a connectionless networking protocol

Uptime The amount of time a system is available and functions

UUID Universally Unique Identifier

V8 JavaScript engine developed by Google

VM Virtual Machine

WebStorage A new standard for letting websites store data on visitors' computers

World of Warcraft A MMORPG game developed by Blizzard. The world's most popular MMORPG by subscriber count.

API Application programming interface

Bibliography

Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, April 2007. URL `http://incubator.apache.org/thrift/static/thrift-20070401.pdf`.

Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, Microelectronics and Information Technology, IMIT, 2003.

Joe Armstrong. Erlang. *Commun. ACM*, 53:68–75, September 2010. ISSN 0001-0782. doi: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1810891.1810910. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1810891.1810910`.

Joe Armstrong. If erlang is the answer, then what is the question?, 2011.

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. doi: http://doi.acm.org/10.1145/1159789.1159792. URL `http://doi.acm.org/10.1145/1159789.1159792`.

Entertainment Software Association. Industry facts, April 2011. URL `http://www.theesa.com/facts/index.asp`.

André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM. ISBN 1-58113-195-X. doi: http://doi.acm.org/10.1145/350391.350432. URL `http://doi.acm.org/10.1145/350391.350432`.

Institute O. Electrical and Electronics E. (ieee). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. 1990.

Johannes Färber. Network game traffic modelling. In *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, pages 53–57, New York, NY, USA, 2002. ACM. ISBN 1-58113-493-2. doi: http://doi.acm.org/10.1145/566500.566508. URL `http://doi.acm.org/10.1145/566500.566508`.

Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31:1–26, March 1999. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/311531.311532. URL `http://doi.acm.org/10.1145/311531.311532`.

Ian Hickson. Web storage – editor's draft 27 april 2011, May 2011. URL `http://dev.w3.org/html5/webstorage/`.

P J Leach and R Salz. Uuids and guids. internet draft draft-leach-uuids-guids-01.txt. internet engineering task force, 1998.

Viktor Lidholt. Design and testing of a generic server for multiplayer gaming. Master's thesis, Uppsala, Sweden, 2002.

Haakan Mattsson, Hans Nilsson, and Claes Wikstrom. Mnesia - a distributed robust dbms for telecommunications applications. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 152–163, London, UK, 1998. Springer-Verlag. ISBN 3-540-65527-1. URL `http://portal.acm.org/citation.cfm?id=645769.667766`.

Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004. ISBN 0201702452.

LLC Nash Information Services. U.s movie market summary 1995 to 2011, April 2011. URL `http://www.the-numbers.com/market/`.

NetHack. Nethack information, April 2011. URL `http://www.nethack.org/common/info.html`.

T. Savor and R. E. Seviora. Hierarchical supervisors for automatic detection of software failures. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, pages 48–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8120-9. URL `http://portal.acm.org/citation.cfm?id=851010.856089`.

Daniel Terdiman. World of warcraft battles server problems. *cnet News*, 04 2006. URL `http://news.cnet.com/World-of-Warcraft-battles-server-problems/2100-1043_3-6063990.html`.