UNIVERSITY OF GOTHENBURG

Reliable Generic Game Server

Niklas Landin
Richard Pannek
Mattias Pettersson
Jonatan Pålsson

Abstract

This is the abstract!

Table of Contents

Chapter 1 Introduction **1**
1.1 Background 2
1.2 Purpose 3
1.3 Challenges in developing the prototype 4
1.4 Limitations of the prototype 4
1.5 Method 5
1.5.1 Development process 5
1.5.2 Design 5
1.5.3 Testing and evaluation 6

Chapter 2 Theory behind the GGS **7**
2.1 Design of the GGS system 7
2.2 Performance 9
2.2.1 Performance measurements 9
2.3 Choice of network protocol 9
2.3.1 HTTP 10
2.3.2 UDP 10
2.3.3 TCP 10
2.4 Generic 10
2.5 Fault Tolerance 10
2.6 Availability 11
2.7 Scalability 11
2.7.1 Load balancing 12
2.7.2 UUID 12
2.8 Security 14
2.8.1 Encryption 14
2.9 Game Development Language in a Virtual Machine 14
2.9.1 JavaScript 14
2.9.2 Other languages 14
2.10 Testing 14

Chapter 3 Implementation of a prototype **15**
3.1 Overview of the prototype 15
3.2 The usage of Erlang in the GGS 17

3.3 Communication with external systems and applications . . . 18
3.4 The modular structure of the GGS prototype . . . 18
3.4.1 The dispatcher module . . . 18
3.4.2 The player module . . . 18
3.4.3 The protocol parser module . . . 19
3.4.4 The coordinator module . . . 19
3.4.5 The table module . . . 19
3.4.6 The game virtual machine module . . . 19
3.4.7 The database module . . . 19
3.5 Techniques for ensuring reliability . . . 19
3.5.1 Supervisor structure . . . 20
3.5.2 Distribution . . . 21
3.5.3 Hot code replacement . . . 21
3.6 Implementation . . . 21
3.7 Example of a GGS server application in Javascript . . . 21

Chapter 4 Problems of implementation . . . 23
4.1 erlang_js . . . 23
4.1.1 UUID . . . 23
4.2 Design choices . . . 23
4.3 Understanding OTP . . . 24
4.4 Usability . . . 24

Chapter 5 Results and discussion . . . 25
5.1 Software development methodology . . . 25
5.2 Statistics . . . 25
5.3 Future improvements . . . 25
5.3.1 Performance . . . 25
5.3.2 Compatibility . . . 25
5.3.3 Setup . . . 25

Chapter 6 Conclusion . . . 27

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. Multi-player games were not as common as they are today, whereas most games today are expected to have a multi-player 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 multi-player capabilities, the demands for multiplayer networking software rises. This thesis is about techniques for improving the quality if this networking software.

The reliable generic game server, hereafter known as 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 inheritly 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 neccessity 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 consistant and available. A consistant and available server is a server that handles

hardware failiures and software failiures 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 where the need for new techniques is large. One specific section where the development has stalled is the game server section. The existing game servers are functional but they lack good fault tolerance and the ability to scale well. Users will notice this in low uptime and many crashes. This is a problem that has existed and been resolved in other industries. In the telecom industry solutins to similar problems have been found.

A common figure often used in telecoms is that of *the nine nines*, referring to 99.999999999% of availability, or rougly $15ms$ 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 telecoms industry. This level of instability should not be accepted in the game server industry either. An unavailabvle phone system could potentially have life threatening consequences, leaving the public unable to contant emergency services. The same can not be said about an unavailable game server. The statement that game servers are less important than phone systems is not a reason not to draw wisdom from what the telecoms have already learnt.

Moving back to the gaming industry. The main reason to develop reliable servers are monetary, it is important for game companies to expand its customer base. Reliable game servers are one improvement that will create a good image of a company. In general the downtime of game servers is much higher than the downtime of telecom system . The structure of the system is similar in many ways and it should be possible to reuse solutions from the telecom system to improve game servers.

In the current state game servers are developed on a per-game basis, in many cases this seems like a bad solution. Developers of network game need to understand network programming. A way to change this is a generic game server which give the game developers a server which they implement their game towards. 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 factors key to the development of GGS have been isolated. Many of these come from the telecom sector. The factors are *scalability, fault tolerance* and being *generic*. 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 defined as the statistical probability of the system functioning as intended at a given point in time. Fault tolerance is defined as the property of a system to always follow a specification, even in the presence of errors. The specification could take the form of 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 of the purposes of this thesis is to investigate how we can 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.

As an aid in discussing the theoretical parts of the GGS a prototype has been developed. The prototype does not feature all of 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 implementation language for the prototype of the GGS was made with inspiration from the telecom industry. The Erlang language was developed by the 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 is also possibly has the largest code base even written in a functional language [Armstrong, 2003]. The same language is used to develop the prototype of the GGS. Usage of Erlang in the GGS is discussed in further detail in section 3.2. Chapter 3 on page 15 provides a description of the prototype developed for this thesis.

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 in order to create a reliable service. The purpose of a reliable game server is to provide a consistant 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 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 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 can not 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, in addition to a broad range of different game types. In order 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 decide 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 to the virtual machine of the GDL from the GGS
- How easy it is to send messages from the GDL VM to the GGS

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 for use in interacting with the GGS. This API is ideally completely independent of the GDL, and reusable for any GDL.

The communication with gaming clients has to take place over 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 large 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, we need a storage platform which is accessible and consistent among all of the GGS, this is also investigated.

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 favour of UDP, due to the fact that the implementation process using TCP was faster 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 9. 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 is one of 50 to 150ms and everything above 200ms is reported to be intolerable 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 (MMORPG:s), 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 between 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 VM:s, the GGS prototype will not support MMORPG:s.

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 are no need to implement more advanced features in the system. It is important to 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

1.5.1 Development process

A prototype was developed early on in the project in order 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 between the modules, they were easily independantly 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 interative workflow, with the first prototype being the first iteration. The second iteration later became the final result of the GGS

1.5.2 Design

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 faulting modules with the last known good data.

1.5.3 Testing and evaluation

Can we use quickcheck?

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, a overview of scalability, fault tolerance and availability is presented.

2.1 Design of the GGS system

The GGS is modelled after a real world system performing much of the same duties as GGS. This is common practice [Armstrong, 2011] in the computer software world, in order to understand complex problems more easily. While there may not always be a real world example of a system performing th exact duties of the system being modelled in the computer, it is often easier to create and analyze requirements for real world systems and processes than systems existing soley 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 following 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 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 to 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 moved 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.

Moves by players are made using the tables present in the chess club. 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.

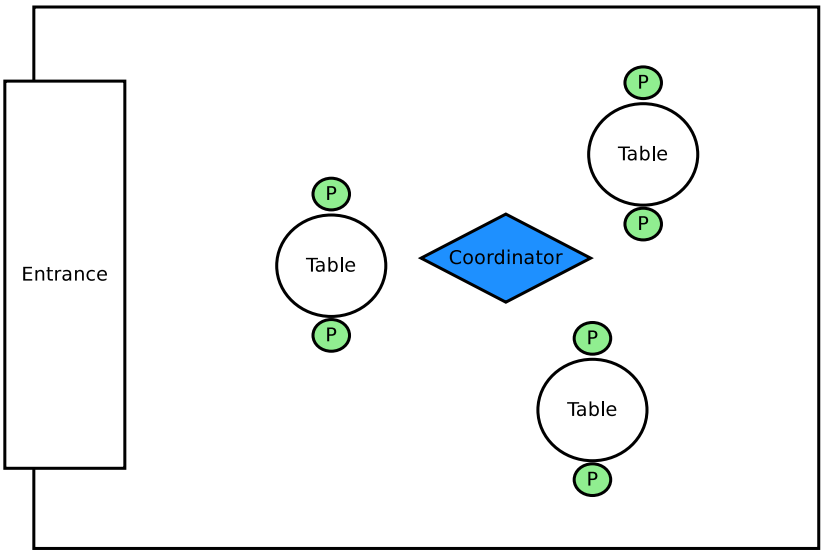

Figure 2.1: The layout of a physical "Chess club" with two players (P) sitting by each chess table (Table), a coordinator keeps track of all moves and players in the building. A player has to pass by the entrance to enter or exit the building. The building is represented by the outermost box.

This isolation of the games plays an imporant part in many properties of the GGS, the isolation means that games can for example be transferred between different chess clubs, furthermore, if cheating takes place, corruption can only occur in the particular table where it was found, and can not spread.

Moving chess players from one location to another is one of the alterations 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 a property 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 GGS can be viewed using algorithm 2.1 on the next page. In order to make this life cycle as efficient and useful as possible, the scalability, fault tolerant and generic traits are added to the GGS. These are not shown in the algorithm, as these traits are tools in making the algorithm behave 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 following page are arbitrary for this example, there are no limits in the GGS on the number of players connecting, for example.

Algorithm 2.1 A very simple example of the flow through the GGS system when a game 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, de-registering 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 is essential. Failiure 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 percieved 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 are discussed in chapter 3 on page 15.

2.2.1 Performance measurements

How many players can we have on a server? Performance differences between games? e.g can one game have thousands players on a server and another only have hundreds? Questions

In the current state game servers is coded for each game that needs it, in many cases this seems like a bad solution. Developers that want to make a network game need to understand network programming. A way to change this is a generic game server which give the game developers a server which they implement their game towards. 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. to be discussed here.

2.3 Choice of network protocol

There are three main ways in which computer communication over the Internet usually takes place; TCP, UDP and HTTP. The first two are transport layer protocols, which are commonly used to transport application layer protocols, such as HTTP. TCP and UDP can not be used on

their own, without an application layer protocol on top. Application layer protocols such as HTTP on the other hand needs a transport layer protocol in order to work.

2.3.1 HTTP

Since HTTP is so widely used on the Internet today in web servers, it is available on most Internet connected devices. This means that if HTTP is used in the GGS, firewalls will not pose problems, 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 makes HTTP unsuitable for our purposes, 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, and what is said about TCP also applies to HTTP.

2.3.2 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 the data 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. In the GGS reliability of transfer was chosen before the speed of the transfer, ruling out UDP as the transport later protocol.

2.3.3 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. In the GGS, data consistency is more important than transfer speeds, and thus TCP is a better alternative than UDP.

2.4 Generic

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 failures. 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(some example here). 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. 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 typ 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 aswell, in brief 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.

Performance penalties

2.6 Availability

One important factor of any server is the availability. A server to which you are unable to connect to is a useless server. Other then within telecomunication, their uptime is of about 99,9999999%, the game developer community hasn't approched this problem very genuinely yet so there is much room for improvement.

There are several good papers on how to migrate whole virtual machines between nodes to replicate them but for the GGS a different approach has been chosen. Instead of just duplicating a virtual machine, the programming language Erlang has been used which offers several features to increase the availability. 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 between 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. With this in mind, the main focus of the GGS is not to move players between tables, but to keep a player in a table, and to start new tables instead. When a server has reached a certain amount 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 between the servers. It is important to investigate and find out how many players that are optimal

Algorithm 2.2 A simple (insufficient) generator for identifiers

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

for each server. This approach makes it possible to utilize 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 between 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.

2.7.2 UUID

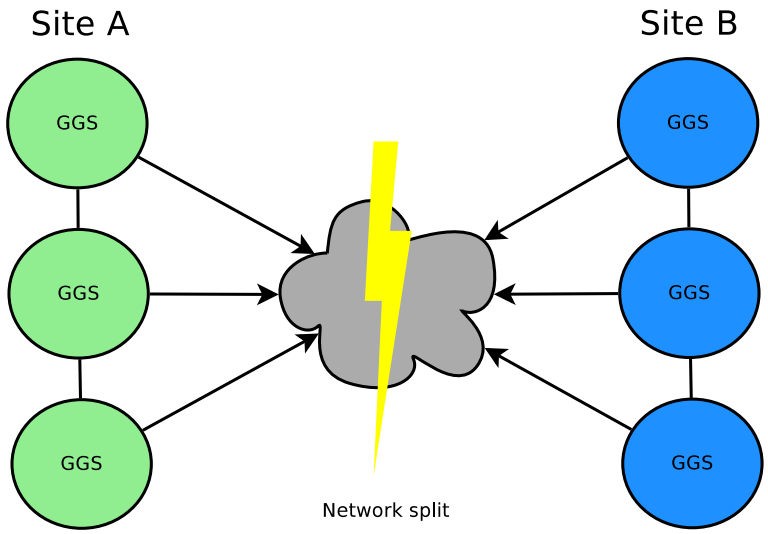

Figure 2.2: An example of a network split

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 ID:s. 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 *Site A* and *Site B* later re-establish communications, they may have generated the same ID:s if using algorithm 2.2, even when mutual system-wide exclusion is implemented. This is exactly the problem UUID:s solve.

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 can not 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 of this not modify the persistent data of other games.

2.8.1 Encryption

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 replacable module which would introduce a 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.

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.

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 <-> JavaScript bridge with our interface.

2.10 Testing

NOTE: This is not a part of the final text. 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.

[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/`

3 Implementation of a prototype

This chapter contains the realization of much of the principles and techniques described in chapter 2 on page 7. 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.

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 devlopment 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 an 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 datastructures related to each process, swapping one process for another (known as *context switching*) is an expensive task in many systems.

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 7) 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 failiure 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 following 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 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 3.4 on page 18. For each connection, a new player process is spawned, which immediately after spawning is integrated in to the GGS by the coordinator process.

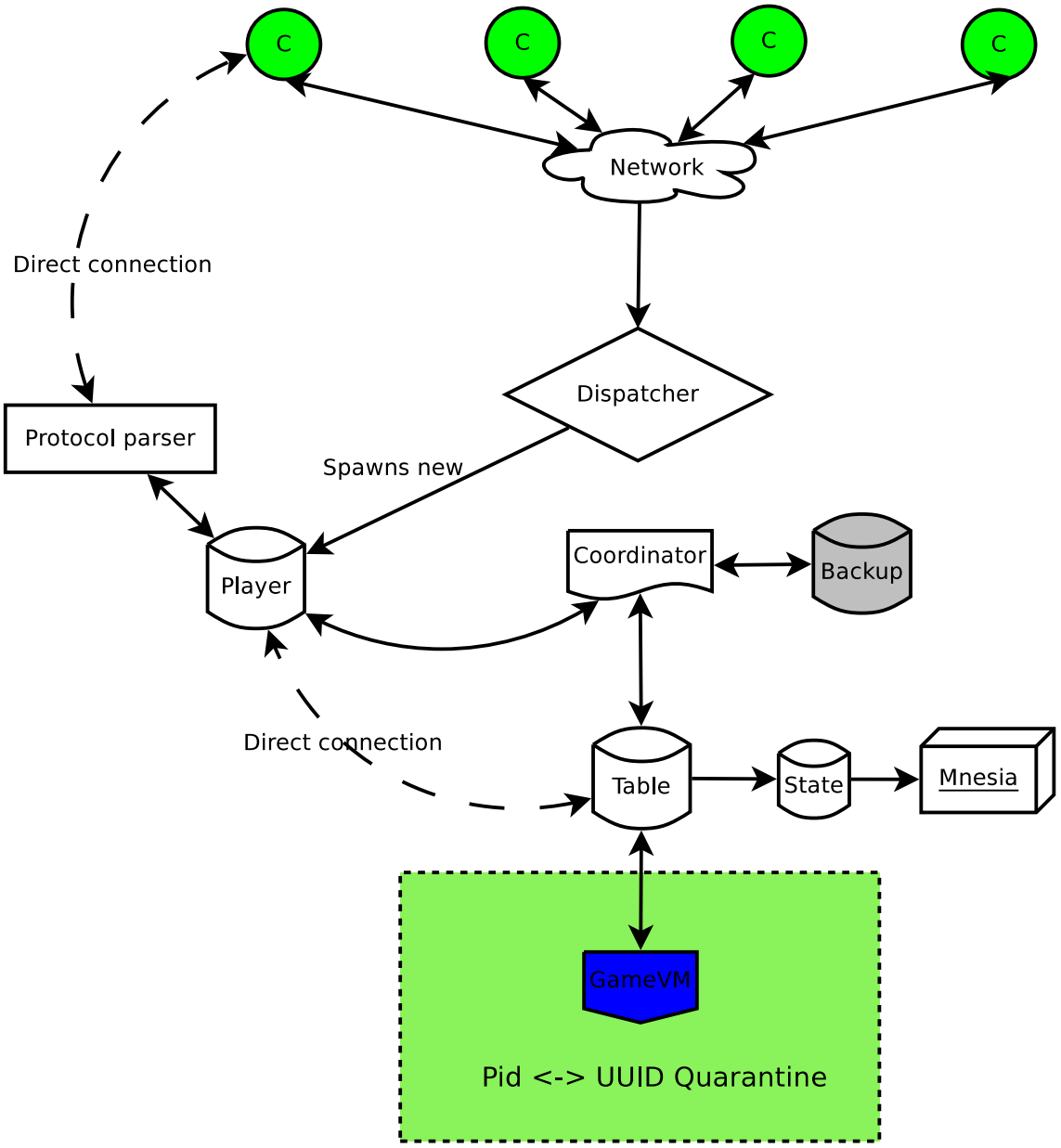

Figure 3.1: The layout of the GGS. The circles marked with 'C' topmost in the picture represent clients. The cloud marked 'network' pictured directly below the clients can be any network, for example the Internet. The barell figure marked 'backup' is a process being fed backup data from the coordinator. 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.

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 instustry to achieve high availability in telecom switches were added to the language.

Erlang uses message passing in favour of shared memory, mutextes 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 favour 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 can not 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 multicore 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 favour 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.* Through ports communication can take place much in the same way communication is performed over sockets.

The GGS uses Erlang ports for generating UUID:s[1] and for interfacing with the virtual machines of games[2].

[1] UUID:s are discussed in section 2.7.2

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

3.3 Communication with external systems and applications

3.4 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 modelling of the GGS after a real world system was discussed in chapter 2 on page 7.

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. Those familiar to object oriented programming may be helped by thinking in the lines of classes and objects.

3.4.1 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 concering 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 23.

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 the club. The actual letting in to the club is in the GGS represented by the creation of a player process discussed in 3.4.2. The newly created player process is handed, and granted rights to, the socket of the newly connected player.

3.4.2 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 3.4.3 on the next page. Raw communication, witout passing the data through a protocol parser is in theory possible, but is not useful.

[3] More information on the SRP is available at:

In the creation of a player process, the coordinator process, discussed in 3.4.4, 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 begind receiving and sending network messaged 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.

3.4.3 The protocol parser module

The protocol parser is an easily interchangable 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 prototocol 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.

The structure of the GGS Protocol

3.4.4 The coordinator module

3.4.5 The table module

3.4.6 The game virtual machine module

3.4.7 The database module

3.5 Techniques for ensuring reliability

One of the main goals of the project is to achieve high reliability. A highly reliable application is one that crashes very, very rarely . 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 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, we can create supervisors. A supervisor is a common approach in ensuring that an application functions in the way it was intended. 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 behaviour 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 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

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

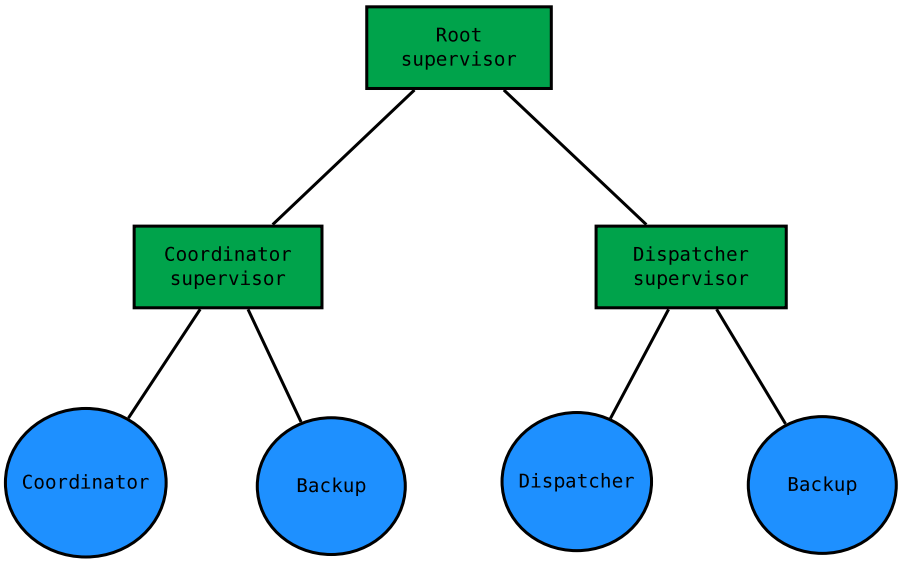

Figure 3.2: The supervisor structure of GGS

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 Distribution

3.5.3 Hot code replacement

3.6 Implementation

User interface

3.7 Example of 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 command from a client, it is passed along to the game VM in a function called *playerCommand.* The *playerCommand* function is responsible for accepting the command into the VM. Typically the *playerCommand* function contains conditional constructs which decide the next action to take. In 3.1 an example of the *playerCommand* function can be seen.

In 3.1 the *playerCommand* function accepts two different commands. The first command is

Algorithm 3.1 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 }
```

a command which allows chat clients connected to the chat server to change nicknames, which are used when chatting. In order to change nickname, a client must send "/nick" immediately followed by a nickname. 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, which is an interface to the database backend contained in the database module (see 3.4.7). The database can be used as any key-value store, however the syntax for insertions and fetch operations is tighty 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 3.3.

4 Problems of implementation

4.1 erlang_js

To be able to run JavaScript on our server we needed to embed a JavaScript engine within the server. After a thorough investigation erlang_js became our choice. Erlang_js provides direct communication with a JavaScript VM (Virtual Machine). This was exactly what we wanted, but we also needed the possibility to communicate from erlang_js to Erlang. This functionality was not yet implemented in erlang_js, due to lack of time.

There were two possible solutions to the problem. We could rewrite some part of erlang_js, or we could switch erlang_js for some other JavaScript engine. Searching for other engines we found erlv8 and beam.js which provided the functionality that we wanted. As we tested beam.js it occurred random crashes of the whole Erlang environment. These crashes were related to the use of erlv8 in beam.js and we decided that the use of erlv8 was not an alternative due to the stability issues.

To get the functionality needed we decided to implement this in erlang_js.

4.1.1 UUID

Erlang identifies processes uniquely throughout the entire Erlang network using process IDs (PID). When we wish to refer to erlang processes from outside our erlang system, for example in a virtual machine for a different language, possibly on a different machine, these PID:s are no longer useful.

This problem is not new, and a common solution is to use a Universally Unique Identifier, a UUID. These identifiers are generated both using randomization and using time. A reasonably large number of UUID:s can be generated before a collision should occur. There are standard tools in many UNIX systems to generate UUID:s, we chose to use the uuidgen command, which employs an equidistributed combined Tausworthe generator.

4.2 Design choices

When designing concurrent applications, it is useful to picture them as real world scenarios, and to model each actor as a real world process. A real world process is a process which performs some action in the real world, such as a mailbox receiving a letter, a door being opened, a person translating a text, a soccer player kicking the ball, just to name a few examples. Since we focus on games in this project, it is suitable to model our system as a place where games take place. We imagined a chess club.

The clients pictured as green circles can be thought of as the physical chess players.

When a player wants to enter the our particular chess club, he must first be let in by the doorman, called the *Dispatcher* in the GGS.

He then gets a name badge, and thus becomes a *Player* process in the system. He is also guided in to the lobby by the *Coordinator*, which has the role of the host of the chess club.

When players wish to play against each other, they talk to the *Coordinator* who pairs them up, and places them at a table. Once they have sat down at the table, they no longer need the assistance of the *Coordinator*, all further communication takes place via the table. This can be thought of as the actual chess game commencing.

All the moves made in the game are recorded by the table, such that the table can restore the game in case something would happen, such as the table tipping over, which would represent the table process crashing.

Once a player wishes to leave a game, or the entire facility, he should contact the *Coordinator*, who revokes his name badge and the *Dispatcher* will let the player out.

With the information kept in the tables and the *Coordinator* combined, we can rebuild the entire state of the server at a different location. This can be thought of the chess club catching fire, and the *Coordinator* rounding up all the tables, running to a new location and building the club up in the exact state it was prior to the fire.

4.3 Understanding OTP

4.4 Usability

5 Results and discussion

5.1 Software development methodology

The project has not followed any specific software development methodology. All work has been based on a predefined schedule and the specifications are made by team members rather than an outside customer or stakeholder. The process can be described as a plan-driven development method going from brainstorming to design, then implementation and finally testing. Yet there have been cycles in the process such as redesign and code refactoring.

5.2 Statistics

5.3 Future improvements

The specification of the GGS prototype is huge and like many other software projects relying on outside technologies, in time it would require a lot of maintanance. Therefore there are a lot of areas in which the GGS could be improved such as performance, compatibility, ease of setup and usage.

5.3.1 Performance

Because of TCP being a connection oriented protocol, it isn't suited for all types of game data transfers. Each transmission will consume more network bandwith than connectionless protocols like UDP and cause uneccessary 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 realtime games as it improves realism and fairness in gameplay and many game developers requires the freedom to take care of safety issues as packet losses themselves. This concludes that UDP would be a benefit for GGS, game developers and game players alike.

5.3.2 Compatibility

GGS relies on modern technologies. This includes the virtual machines(vm) that the GGS uses for communication between Erlang and the GDL:s. These specific vm:s are crucial for game developers to be able to write games in other languages than Erlang. Therefore it would be best for the GGS to have as many of these vm:s implemented as possible. The vm:s taken into consideration so far have been unstable or incomplete and it is possible to search for more vm:s, testing them and intergrate them into the GGS prototype.

5.3.3 Setup

The GGS prototype installation procedure requires different configuring and building steps and thus it isn't in an acceptable release state. An executable installation file for each supported platform would be optimal.

5.3.4 Usage

The GGS doesn't support many programming languages nor does it have a complete documentation. This needs to be taken care of in future versions.

6 Conclusion

Glossary

Here we could add some important words and their definitions..

Bibliography

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.

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`.

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`.

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.

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`.