Generic Game Server

Jonatan Pålsson

Niklas Landin

Richard Pannek

Matias Petterson

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

1.2 Purpose

1.3 Challenges

Challenges lies mainly in providing a reliable, high-performing server and at the same time make it easy to use for game developers.

1.3.1 Basis

1.4 Delimitations

1.4.1 Types of games

In theory no limitations, but in reality it will be limitations. Many factors are involved here. Implementation of protocol, storage possibilities, server capacity, language support. In real time games a low latency is very important not a high bandwidth because the games already send very little data, ~ 80 bytes. Lag of below 250 ms is good, lag up to 500 ms payable and beyond that the lag is noticeable.

1.5 Method

1.5.1 Development process

May be Extreme Programming(XP), need to check this out further. Maybe adapt so we can say that we use a standardized software development method.

1.5.1.1 Demand specification

1.5.2 Design

1.5.3 Testing and evaluation

Can we use quickcheck?

Chapter 2

Theory

2.0.4 Performance

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 to be discussed here.

2.0.5 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.

2.0.5.1 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 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 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.

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

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

2.0.6 Encryption

2.0.6.1 Performance penalties

2.0.7 Availability

One important factor of a server is the availability, a server that you can not connect to is a bad server. Erlang has several features to increase the availability, for example hot code replacement. It is also critical to have a good design, we want to separate each part of the server and thus avoiding that the whole server will crash.

2.0.8 Scalability

Because P2P game architectures are a constant goal for cheaters and because “Cheating is a major concern in network games as it degrades the experience of the majority of players who are honest” and preventing cheating in P2P game architectures is very difficult game developers try to use Client - Server architectures which have a natural problem to scale. In this paper we want to show some strategies to achieve scalability.

2.0.8.1 UUID

2.0.9 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.

Chapter 3

Overview

3.0.10 Techniques for ensuring reliability

One of the main goals of the project is to achieve high reliability. A highly reliable application is one 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.

3.0.10.1 Supervisor structure

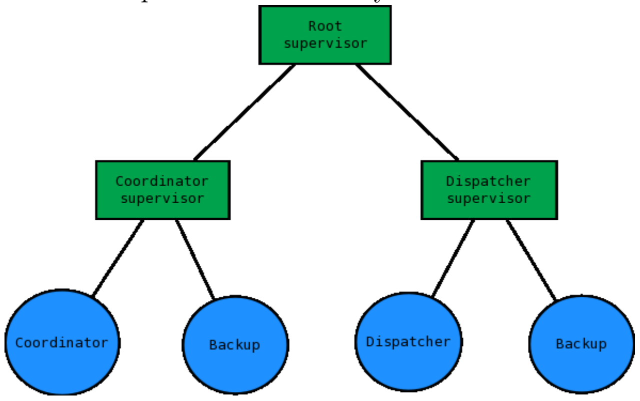
By linking processes together and notifying parents when children exit, we can create supervisors. A supervisor is a common approach in ensuring that an application functions in the way it was intended. When a process misbehaves, the supervisor takes some action to restore the process to a functional 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 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 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.

3.0.10.2 Hot code replacement

3.0.11 Implementation

3.0.11.1 User interface

Chapter 4

Problems

4.0.12 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.

4.0.12.1 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 PIDs 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 UUIDs can be generated before a collision should occur. There are standard tools in many UNIX systems to generate UUIDs, we chose to use the uuidgen command, which employs an equidistributed combined Tausworthe generator.

4.1 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 GGS.

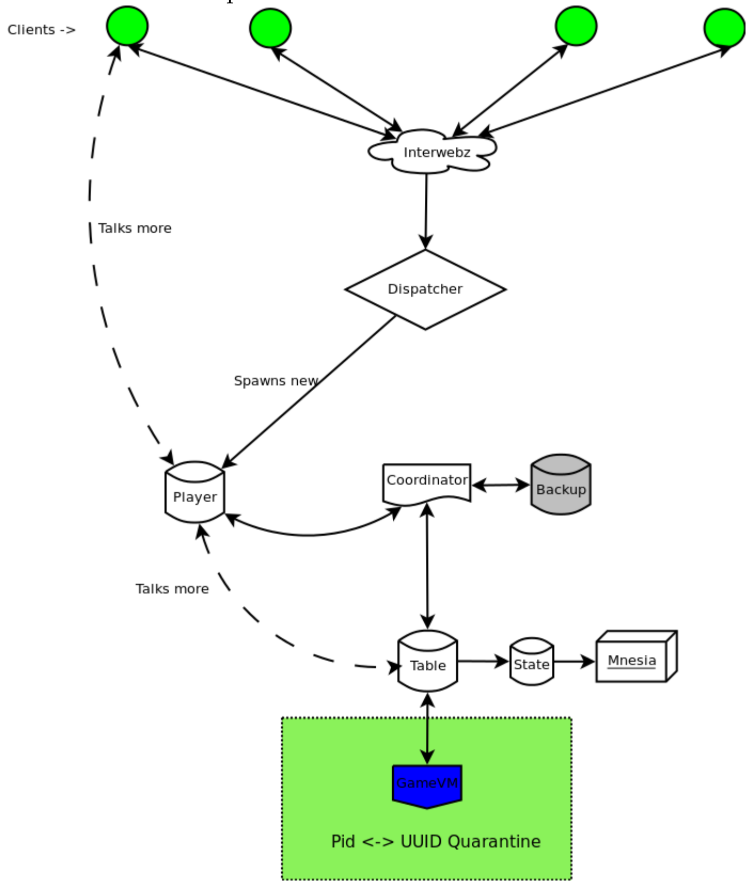
He then gets a name badge, and thus becomes a Player process in the system. He is also guided in to the lobby by the Coordinator, which has the role of the host of the chess club.

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.



4.2 Understanding OTP

4.3 Usability

Chapter 5

Results and discussion

5.1 Software development methodology

5.2 Statistics

Chapter 6

Conclusion

Chapter 7

References

Chapter 8

Appendix

Text goes here..

Bibliography

[1] Savor, T.; Seviora, R.E.; , "Hierarchical supervisors for automatic detection of software failures," PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering , vol., no., pp.48-59, 2-5 Nov1997 doi: 10.1109/ISSRE.1997.630847 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=630847&isnumber=13710

[2] Vinoski, S.; , "Reliability with Erlang," Internet Computing, IEEE , vol.11, no.6, pp.79-81, Nov.-Dec. 2007 doi: 10.1109/MIC.2007.132 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4376232&isnumber=4376216

[3] CESARINI, F., & THOMPSON, S. (2009). Erlang programming. Beijing, O'Reilly. pp.139

[4] "Erlang/OTP Product Information: Technical Description of Erlang." Home of Erlang/OTP. Web. 01 Mar. 2011. <http://www.erlang.se/productinfo/erlang_tech.shtml>.

[5] Joe Armstrong – Armstrong, J. [2011]. If Erlang is the answer, then what is the question?. [1]. IT University. Computer Science and Engineering, 15/2/2011

[6] Gul Abdulnabi Agha (1985). ACTORS: A MODEL OF CONCURRENT COMPUTATION IN DISTRIBUTED SYSTEMS. Ph.D thesis, Artificial Intelligence Laboratory, MIT.